Spring Security Reference

Ben Alex, Luke Taylor, Rob Winch, Gunnar Hillert, Joe Grandja, Jay Bryant, Eddú Meléndez, Josh Cummings, Dave Syer, Eleftheria Stein

Version 5.6.0-M1

Table of Contents

Spring Security is a framework that provides authentication, authorization, and protection against common attacks. With first class support for both imperative and reactive applications, it is the de-facto standard for securing Spring-based applications.

Introduction

This section discusses the logistics of Spring Security.

Chapter 1. Prerequisites

Spring Security requires a Java 8 or higher Runtime Environment.

As Spring Security aims to operate in a self-contained manner, you do not need to place any special configuration files in your Java Runtime Environment. In particular, you need not configure a special Java Authentication and Authorization Service (JAAS) policy file or place Spring Security into common classpath locations.

Similarly, if you use an EJB Container or Servlet Container, you need not put any special configuration files anywhere nor include Spring Security in a server classloader. All the required files are contained within your application.

This design offers maximum deployment time flexibility, as you can copy your target artifact (be it a JAR, WAR, or EAR) from one system to another and it immediately works.

Chapter 2. Spring Security Community

Welcome to the Spring Security Community! This section discusses how you can make the most of our vast community.

2.1. Getting Help

If you need help with Spring Security, we are here to help. The following are some of the best ways to get help:

- Read through this documentation.
- Try one of our many sample applications.
- Ask a question on https://stackoverflow.com with the `spring-security` tag.
- Report bugs and enhancement requests at https://github.com/spring-projects/spring-security/issues

2.2. Becoming Involved

We welcome your involvement in the Spring Security project. There are many ways to contribute, including answering questions on Stack Overflow, writing new code, improving existing code, assisting with documentation, developing samples or tutorials, reporting bugs, or simply making suggestions. For more information, see our Contributing documentation.

2.3. Source Code

You can find Spring Security's source code on GitHub at https://github.com/spring-projects/spring-security/

2.4. Apache 2 License

Spring Security is Open Source software released under the Apache 2.0 license.

2.5. Social Media

You can follow @SpringSecurity and the Spring Security team on Twitter to stay up to date with the latest news. You can also follow @SpringCentral to keep up to date with the entire Spring portfolio.

Chapter 3. What's New in Spring Security 5.6

Spring Security 5.6 provides a number of new features. Below are the highlights of the release.

3.1. Servlet

- Configuration
  - Introduced `AuthorizationManager` for method security

Chapter 4. Getting Spring Security

This section discusses all you need to know about getting the Spring Security binaries. See Source Code for how to obtain the source code.

4.1. Release Numbering

Spring Security versions are formatted as MAJOR.MINOR.PATCH such that:

- MAJOR versions may contain breaking changes. Typically, these are done to provide improved security to match modern security practices.
- MINOR versions contain enhancements but are considered passive updates
- PATCH level should be perfectly compatible, forwards and backwards, with the possible exception of changes that fix bugs.

4.2. Usage with Maven

As most open source projects, Spring Security deploys its dependencies as Maven artifacts. The topics in this section provide detail on how to consume Spring Security when using Maven.

4.2.1. Spring Boot with Maven

Spring Boot provides a `spring-boot-starter-security` starter that aggregates Spring Security-related dependencies together. The simplest and preferred way to use the starter is to use Spring Initializr by using an IDE integration (Eclipse, IntelliJ, NetBeans) or through https://start.spring.io.

Alternatively, you can manually add the starter, as the following example shows:

Example 1. pom.xml

```
<dependencies>
    <!-- ... other dependency elements ... -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
</dependencies>
```

Since Spring Boot provides a Maven BOM to manage dependency versions, you do not need to specify a version. If you wish to override the Spring Security version, you may do so by providing a Maven property, as the following example shows:

Example 2. pom.xml

```
<properties>
    <!-- ... -->
    <spring-security.version>5.6.0-M1</spring-security.version>
</dependencies>
```

Since Spring Security makes breaking changes only in major releases, it is safe to use a newer version of Spring Security with Spring Boot. However, at times, you may need to update the version of Spring Framework as well. You can do so by adding a Maven property, as the following example shows:

Example 3. pom.xml

```
<properties>
    <!-- ... -->
    <spring.version>5.3.9</spring.version>
</dependencies>
```

If you use additional features (such as LDAP, OpenID, and others), you need to also include the appropriate Project Modules and Dependencies.

4.2.2. Maven Without Spring Boot

When you use Spring Security without Spring Boot, the preferred way is to use Spring Security's BOM to ensure a consistent version of Spring Security is used throughout the entire project. The following example shows how to do so:

Example 4. pom.xml

```
<dependencyManagement>
    <dependencies>
        <!-- ... other dependency elements ... -->
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-bom</artifactId>
            <version>{spring-security-version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

A minimal Spring Security Maven set of dependencies typically looks like the following:

Example 5. pom.xml

```
<dependencies>
    <!-- ... other dependency elements ... -->
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-config</artifactId>
    </dependency>
</dependencies>
```

If you use additional features (such as LDAP, OpenID, and others), you need to also include the appropriate Project Modules and Dependencies.

Spring Security builds against Spring Framework 5.3.9 but should generally work with any newer version of Spring Framework 5.x. Many users are likely to run afoul of the fact that Spring Security's transitive dependencies resolve Spring Framework 5.3.9, which can cause strange classpath problems. The easiest way to resolve this is to use the `spring-framework-bom` within the `<dependencyManagement>` section of your `pom.xml` as the following example shows:

Example 6. pom.xml

```
<dependencyManagement>
    <dependencies>
        <!-- ... other dependency elements ... -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-framework-bom</artifactId>
            <version>5.3.9</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

The preceding example ensures that all the transitive dependencies of Spring Security use the Spring 5.3.9 modules.

This approach uses Maven's "bill of materials" (BOM) concept and is only available in Maven 2.0.9+. For additional details about how dependencies are resolved, see Maven's Introduction to the Dependency Mechanism documentation.

4.2.3. Maven Repositories

All GA releases (that is, versions ending in .RELEASE) are deployed to Maven Central, so no additional Maven repositories need to be declared in your pom.

If you use a SNAPSHOT version, you need to ensure that you have the Spring Snapshot repository defined, as the following example shows:

Example 7. pom.xml

```
<repositories>
    <!-- ... possibly other repository elements ... -->
    <repository>
        <id>spring-snapshot</id>
        <name>Spring Snapshot Repository</name>
        <url>https://repo.spring.io/snapshot</url>
    </repository>
</repositories>
```

If you use a milestone or release candidate version, you need to ensure that you have the Spring Milestone repository defined, as the following example shows:

Example 8. pom.xml

```
<repositories>
    <!-- ... possibly other repository elements ... -->
    <repository>
        <id>spring-milestone</id>
        <name>Spring Milestone Repository</name>
        <url>https://repo.spring.io/milestone</url>
    </repository>
</repositories>
```

4.3. Gradle

As most open source projects, Spring Security deploys its dependencies as Maven artifacts, which allows for first-class Gradle support. The following topics provide detail on how to consume Spring Security when using Gradle.

4.3.1. Spring Boot with Gradle

Spring Boot provides a `spring-boot-starter-security` starter that aggregates Spring Security related dependencies together. The simplest and preferred method to use the starter is to use Spring Initializr by using an IDE integration (Eclipse, IntelliJ, NetBeans) or through https://start.spring.io.

Alternatively, you can manually add the starter, as the following example shows:

Example 9. build.gradle

```
dependencies {
    compile "org.springframework.boot:spring-boot-starter-security"
}
```

Since Spring Boot provides a Maven BOM to manage dependency versions, you need not specify a version. If you wish to override the Spring Security version, you may do so by providing a Gradle property, as the following example shows:

Example 10. build.gradle

```
ext['spring-security.version']='5.6.0-M1'
```

Since Spring Security makes breaking changes only in major releases, it is safe to use a newer version of Spring Security with Spring Boot. However, at times, you may need to update the version of Spring Framework as well. You can do so by adding a Gradle property, as the following example shows:

Example 11. build.gradle

```
ext['spring.version']='5.3.9'
```

If you use additional features (such as LDAP, OpenID, and others), you need to also include the appropriate Project Modules and Dependencies.

4.3.2. Gradle Without Spring Boot

When you use Spring Security without Spring Boot, the preferred way is to use Spring Security's BOM to ensure a consistent version of Spring Security is used throughout the entire project. You can do so by using the Dependency Management Plugin, as the following example shows:

Example 12. build.gradle

```
plugins {
    id "io.spring.dependency-management" version "1.0.6.RELEASE"
}

dependencyManagement {
    imports {
        mavenBom 'org.springframework.security:spring-security-bom:5.6.0-M1'
    }
}
```

A minimal Spring Security Maven set of dependencies typically looks like the following:

Example 13. build.gradle

```
dependencies {
    compile "org.springframework.security:spring-security-web"
    compile "org.springframework.security:spring-security-config"
}
```

If you use additional features (such as LDAP, OpenID, and others), you need to also include the appropriate Project Modules and Dependencies.

Spring Security builds against Spring Framework 5.3.9 but should generally work with any newer version of Spring Framework 5.x. Many users are likely to run afoul of the fact that Spring Security's transitive dependencies resolve Spring Framework 5.3.9, which can cause strange classpath problems. The easiest way to resolve this is to use the `spring-framework-bom` within your `<dependencyManagement>` section of your `pom.xml`. You can do so by using the Dependency Management Plugin, as the following example shows:

Example 14. build.gradle

```
plugins {
    id "io.spring.dependency-management" version "1.0.6.RELEASE"
}

dependencyManagement {
    imports {
        mavenBom 'org.springframework:spring-framework-bom:5.3.9'
    }
}
```

The preceding example ensures that all the transitive dependencies of Spring Security use the

Spring 5.3.9 modules.

4.3.3. Gradle Repositories

All GA releases (that is, versions ending in .RELEASE) are deployed to Maven Central, so using the mavenCentral() repository is sufficient for GA releases. The following example shows how to do so:

Example 15. build.gradle

```
repositories {
    mavenCentral()
}
```

If you use a SNAPSHOT version, you need to ensure you have the Spring Snapshot repository defined, as the following example shows:

Example 16. build.gradle

```
repositories {
    maven { url 'https://repo.spring.io/snapshot' }
}
```

If you use a milestone or release candidate version, you need to ensure that you have the Spring Milestone repository defined, as the following example shows:

Example 17. build.gradle

```
repositories {
    maven { url 'https://repo.spring.io/milestone' }
}
```

Chapter 5. Features

Spring Security provides comprehensive support for authentication, authorization, and protection against common exploits. It also provides integration with other libraries to simplify its usage.

5.1. Authentication

Spring Security provides comprehensive support for authentication. Authentication is how we verify the identity of who is trying to access a particular resource. A common way to authenticate users is by requiring the user to enter a username and password. Once authentication is performed we know the identity and can perform authorization.

5.1.1. Authentication Support

Spring Security provides built in support for authenticating users. Refer to the sections on authentication for Servlet and WebFlux for details on what is supported for each stack.

5.1.2. Password Storage

Spring Security's `PasswordEncoder` interface is used to perform a one way transformation of a password to allow the password to be stored securely. Given `PasswordEncoder` is a one way transformation, it is not intended when the password transformation needs to be two way (i.e. storing credentials used to authenticate to a database). Typically `PasswordEncoder` is used for storing a password that needs to be compared to a user provided password at the time of authentication.

Password Storage History

Throughout the years the standard mechanism for storing passwords has evolved. In the beginning passwords were stored in plain text. The passwords were assumed to be safe because the data store the passwords were saved in required credentials to access it. However, malicious users were able to find ways to get large "data dumps" of usernames and passwords using attacks like SQL Injection. As more and more user credentials became public security experts realized we needed to do more to protect users' passwords.

Developers were then encouraged to store passwords after running them through a one way hash such as SHA-256. When a user tried to authenticate, the hashed password would be compared to the hash of the password that they typed. This meant that the system only needed to store the one way hash of the password. If a breach occurred, then only the one way hashes of the passwords were exposed. Since the hashes were one way and it was computationally difficult to guess the passwords given the hash, it would not be worth the effort to figure out each password in the system. To defeat this new system malicious users decided to create lookup tables known as Rainbow Tables. Rather than doing the work of guessing each password every time, they computed the password once and stored it in a lookup table.

To mitigate the effectiveness of Rainbow Tables, developers were encouraged to use salted passwords. Instead of using just the password as input to the hash function, random bytes (known as salt) would be generated for every users' password. The salt and the user's password would be ran through the hash function which produced a unique hash. The salt would be stored alongside

the user's password in clear text. Then when a user tried to authenticate, the hashed password would be compared to the hash of the stored salt and the password that they typed. The unique salt meant that Rainbow Tables were no longer effective because the hash was different for every salt and password combination.

In modern times we realize that cryptographic hashes (like SHA-256) are no longer secure. The reason is that with modern hardware we can perform billions of hash calculations a second. This means that we can crack each password individually with ease.

Developers are now encouraged to leverage adaptive one-way functions to store a password. Validation of passwords with adaptive one-way functions are intentionally resource (i.e. CPU, memory, etc) intensive. An adaptive one-way function allows configuring a "work factor" which can grow as hardware gets better. It is recommended that the "work factor" be tuned to take about 1 second to verify a password on your system. This trade off is to make it difficult for attackers to crack the password, but not so costly it puts excessive burden on your own system. Spring Security has attempted to provide a good starting point for the "work factor", but users are encouraged to customize the "work factor" for their own system since the performance will vary drastically from system to system. Examples of adaptive one-way functions that should be used include bcrypt, PBKDF2, scrypt, and argon2.

Because adaptive one-way functions are intentionally resource intensive, validating a username and password for every request will degrade performance of an application significantly. There is nothing Spring Security (or any other library) can do to speed up the validation of the password since security is gained by making the validation resource intensive. Users are encouraged to exchange the long term credentials (i.e. username and password) for a short term credential (i.e. session, OAuth Token, etc). The short term credential can be validated quickly without any loss in security.

DelegatingPasswordEncoder

Prior to Spring Security 5.0 the default `PasswordEncoder` was `NoOpPasswordEncoder` which required plain text passwords. Based upon the Password History section you might expect that the default `PasswordEncoder` is now something like `BCryptPasswordEncoder`. However, this ignores three real world problems:

- There are many applications using old password encodings that cannot easily migrate
- The best practice for password storage will change again.
- As a framework Spring Security cannot make breaking changes frequently

Instead Spring Security introduces `DelegatingPasswordEncoder` which solves all of the problems by:

- Ensuring that passwords are encoded using the current password storage recommendations
- Allowing for validating passwords in modern and legacy formats
- Allowing for upgrading the encoding in the future

You can easily construct an instance of `DelegatingPasswordEncoder` using `PasswordEncoderFactories`.

Example 18. Create Default DelegatingPasswordEncoder

Java

```
PasswordEncoder passwordEncoder =
    PasswordEncoderFactories.createDelegatingPasswordEncoder();
```

Kotlin

```
val passwordEncoder: PasswordEncoder =
PasswordEncoderFactories.createDelegatingPasswordEncoder()
```

Alternatively, you may create your own custom instance. For example:

Example 19. Create Custom DelegatingPasswordEncoder

Java

```
String idForEncode = "bcrypt";
Map encoders = new HashMap<>();
encoders.put(idForEncode, new BCryptPasswordEncoder());
encoders.put("noop", NoOpPasswordEncoder.getInstance());
encoders.put("pbkdf2", new Pbkdf2PasswordEncoder());
encoders.put("scrypt", new SCryptPasswordEncoder());
encoders.put("sha256", new StandardPasswordEncoder());

PasswordEncoder passwordEncoder =
    new DelegatingPasswordEncoder(idForEncode, encoders);
```

Kotlin

```
val idForEncode = "bcrypt"
val encoders: MutableMap<String, PasswordEncoder> = mutableMapOf()
encoders[idForEncode] = BCryptPasswordEncoder()
encoders["noop"] = NoOpPasswordEncoder.getInstance()
encoders["pbkdf2"] = Pbkdf2PasswordEncoder()
encoders["scrypt"] = SCryptPasswordEncoder()
encoders["sha256"] = StandardPasswordEncoder()

val passwordEncoder: PasswordEncoder = DelegatingPasswordEncoder(idForEncode,
encoders)
```

Password Storage Format

The general format for a password is:

Example 20. DelegatingPasswordEncoder Storage Format

```
{id}encodedPassword
```

Such that `id` is an identifier used to look up which `PasswordEncoder` should be used and `encodedPassword` is the original encoded password for the selected `PasswordEncoder`. The `id` must be at the beginning of the password, start with `{` and end with `}`. If the `id` cannot be found, the `id` will be null. For example, the following might be a list of passwords encoded using different `id`. All of the original passwords are "password".

Example 21. DelegatingPasswordEncoder Encoded Passwords Example

```
{bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG ①
{noop}password ②
{pbkdf2}5d923b44a6d129f3ddf3e3c8d29412723dcbde72445e8ef6bf3b508fbf17fa4ed4d6b99ca7
63d8dc ③
{scrypt}$e0801$8bWJaSu2IKSn9Z9kM+TPXfOc/9bdYSrN1oD9qfVThWEwdRTnO7re7Ei+fUZRJ68k9lT
yuTeUp4of4g24hHnazw==$OAOec05+bXxvuu/1qZ6NUR+xQYvYv7BeL1QxwRpY5Pc= ④
{sha256}97cde38028ad898ebc02e690819fa220e88c62e0699403e94fff291cfffaf8410849f27605
abcbc0 ⑤
```

① The first password would have a `PasswordEncoder` id of `bcrypt` and encodedPassword of `$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG`. When matching it would delegate to `BCryptPasswordEncoder`

② The second password would have a `PasswordEncoder` id of `noop` and encodedPassword of `password`. When matching it would delegate to `NoOpPasswordEncoder`

③ The third password would have a `PasswordEncoder` id of `pbkdf2` and encodedPassword of `5d923b44a6d129f3ddf3e3c8d29412723dcbde72445e8ef6bf3b508fbf17fa4ed4d6b99ca763d8dc`. When matching it would delegate to `Pbkdf2PasswordEncoder`

④ The fourth password would have a `PasswordEncoder` id of `scrypt` and encodedPassword of `$e0801$8bWJaSu2IKSn9Z9kM+TPXfOc/9bdYSrN1oD9qfVThWEwdRTnO7re7Ei+fUZRJ68k9lTyuTeUp4of4g24hHnazw==$OAOec05+bXxvuu/1qZ6NUR+xQYvYv7BeL1QxwRpY5Pc=` When matching it would delegate to `SCryptPasswordEncoder`

⑤ The final password would have a `PasswordEncoder` id of `sha256` and encodedPassword of `97cde38028ad898ebc02e690819fa220e88c62e0699403e94fff291cfffaf8410849f27605abcbc0`. When matching it would delegate to `StandardPasswordEncoder`

Some users might be concerned that the storage format is provided for a potential hacker. This is not a concern because the storage of the password does not rely on the algorithm being a secret. Additionally, most formats are easy for an attacker to figure out without the prefix. For example, BCrypt passwords often start with `$2a$`.

Password Encoding

The `idForEncode` passed into the constructor determines which `PasswordEncoder` will be used for encoding passwords. In the `DelegatingPasswordEncoder` we constructed above, that means that the result of encoding `password` would be delegated to `BCryptPasswordEncoder` and be prefixed with `{bcrypt}`. The end result would look like:

Example 22. DelegatingPasswordEncoder Encode Example

```
{bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG
```

Password Matching

Matching is done based upon the `{id}` and the mapping of the `id` to the `PasswordEncoder` provided in the constructor. Our example in Password Storage Format provides a working example of how this is done. By default, the result of invoking `matches(CharSequence, String)` with a password and an `id` that is not mapped (including a null id) will result in an `IllegalArgumentException`. This behavior can be customized using `DelegatingPasswordEncoder.setDefaultPasswordEncoderForMatches(PasswordEncoder)`.

By using the `id` we can match on any password encoding, but encode passwords using the most modern password encoding. This is important, because unlike encryption, password hashes are designed so that there is no simple way to recover the plaintext. Since there is no way to recover the plaintext, it makes it difficult to migrate the passwords. While it is simple for users to migrate `NoOpPasswordEncoder`, we chose to include it by default to make it simple for the getting started experience.

Getting Started Experience

If you are putting together a demo or a sample, it is a bit cumbersome to take time to hash the passwords of your users. There are convenience mechanisms to make this easier, but this is still not intended for production.

Example 23. withDefaultPasswordEncoder Example

Java

```
User user = User.withDefaultPasswordEncoder()
  .username("user")
  .password("password")
  .roles("user")
  .build();
System.out.println(user.getPassword());
// {bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG
```

Kotlin

```
val user = User.withDefaultPasswordEncoder()
    .username("user")
    .password("password")
    .roles("user")
    .build()
println(user.password)
// {bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG
```

If you are creating multiple users, you can also reuse the builder.

Example 24. withDefaultPasswordEncoder Reusing the Builder

Java

```
UserBuilder users = User.withDefaultPasswordEncoder();
User user = users
  .username("user")
  .password("password")
  .roles("USER")
  .build();
User admin = users
  .username("admin")
  .password("password")
  .roles("USER","ADMIN")
  .build();
```

Kotlin

```
val users = User.withDefaultPasswordEncoder()
val user = users
    .username("user")
    .password("password")
    .roles("USER")
    .build()
val admin = users
    .username("admin")
    .password("password")
    .roles("USER", "ADMIN")
    .build()
```

This does hash the password that is stored, but the passwords are still exposed in memory and in the compiled source code. Therefore, it is still not considered secure for a production environment. For production, you should hash your passwords externally.

Encode with Spring Boot CLI

The easiest way to properly encode your password is to use the Spring Boot CLI.

For example, the following will encode the password of `password` for use with DelegatingPasswordEncoder:

Example 25. Spring Boot CLI encodepassword Example

```
spring encodepassword password
{bcrypt}$2a$10$X5wFBtLrL/kHcmrOGGTrGufsBX8CJ0WpQpF3pgeuxBB/H73BK1DW6
```

Troubleshooting

The following error occurs when one of the passwords that are stored has no id as described in Password Storage Format.

```
java.lang.IllegalArgumentException: There is no PasswordEncoder mapped for the id
"null"
    at
org.springframework.security.crypto.password.DelegatingPasswordEncoder$UnmappedIdPassw
ordEncoder.matches(DelegatingPasswordEncoder.java:233)
    at
org.springframework.security.crypto.password.DelegatingPasswordEncoder.matches(Delegat
ingPasswordEncoder.java:196)
```

The easiest way to resolve the error is to switch to explicitly provide the `PasswordEncoder` that you passwords are encoded with. The easiest way to resolve it is to figure out how your passwords are currently being stored and explicitly provide the correct `PasswordEncoder`.

If you are migrating from Spring Security 4.2.x you can revert to the previous behavior by exposing a `NoOpPasswordEncoder` bean.

Alternatively, you can prefix all of your passwords with the correct id and continue to use `DelegatingPasswordEncoder`. For example, if you are using BCrypt, you would migrate your password from something like:

```
$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG
```

to

```
{bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG
```

For a complete listing of the mappings refer to the Javadoc on PasswordEncoderFactories.

BCryptPasswordEncoder

The `BCryptPasswordEncoder` implementation uses the widely supported bcrypt algorithm to hash the passwords. In order to make it more resistent to password cracking, bcrypt is deliberately slow. Like other adaptive one-way functions, it should be tuned to take about 1 second to verify a password on your system. The default implementation of `BCryptPasswordEncoder` uses strength 10 as mentioned in the Javadoc of BCryptPasswordEncoder. You are encouraged to tune and test the strength parameter on your own system so that it takes roughly 1 second to verify a password.

Example 26. BCryptPasswordEncoder

Java

```
// Create an encoder with strength 16
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(16);
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

Kotlin

```
// Create an encoder with strength 16
val encoder = BCryptPasswordEncoder(16)
val result: String = encoder.encode("myPassword")
assertTrue(encoder.matches("myPassword", result))
```

Argon2PasswordEncoder

The `Argon2PasswordEncoder` implementation uses the Argon2 algorithm to hash the passwords. Argon2 is the winner of the Password Hashing Competition. In order to defeat password cracking on custom hardware, Argon2 is a deliberately slow algorithm that requires large amounts of memory. Like other adaptive one-way functions, it should be tuned to take about 1 second to verify a password on your system. The current implementation of the `Argon2PasswordEncoder` requires BouncyCastle.

Example 27. Argon2PasswordEncoder

Java

```
// Create an encoder with all the defaults
Argon2PasswordEncoder encoder = new Argon2PasswordEncoder();
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

Kotlin

```
// Create an encoder with all the defaults
val encoder = Argon2PasswordEncoder()
val result: String = encoder.encode("myPassword")
assertTrue(encoder.matches("myPassword", result))
```

Pbkdf2PasswordEncoder

The `Pbkdf2PasswordEncoder` implementation uses the PBKDF2 algorithm to hash the passwords. In order to defeat password cracking PBKDF2 is a deliberately slow algorithm. Like other adaptive one-way functions, it should be tuned to take about 1 second to verify a password on your system.

This algorithm is a good choice when FIPS certification is required.

Example 28. Pbkdf2PasswordEncoder

Java

```
// Create an encoder with all the defaults
Pbkdf2PasswordEncoder encoder = new Pbkdf2PasswordEncoder();
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

Kotlin

```
// Create an encoder with all the defaults
val encoder = Pbkdf2PasswordEncoder()
val result: String = encoder.encode("myPassword")
assertTrue(encoder.matches("myPassword", result))
```

SCryptPasswordEncoder

The `SCryptPasswordEncoder` implementation uses scrypt algorithm to hash the passwords. In order to defeat password cracking on custom hardware scrypt is a deliberately slow algorithm that requires large amounts of memory. Like other adaptive one-way functions, it should be tuned to take about 1 second to verify a password on your system.

Example 29. SCryptPasswordEncoder

Java

```
// Create an encoder with all the defaults
SCryptPasswordEncoder encoder = new SCryptPasswordEncoder();
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

Kotlin

```
// Create an encoder with all the defaults
val encoder = SCryptPasswordEncoder()
val result: String = encoder.encode("myPassword")
assertTrue(encoder.matches("myPassword", result))
```

Other PasswordEncoders

There are a significant number of other `PasswordEncoder` implementations that exist entirely for backward compatibility. They are all deprecated to indicate that they are no longer considered secure. However, there are no plans to remove them since it is difficult to migrate existing legacy

systems.

Password Storage Configuration

Spring Security uses DelegatingPasswordEncoder by default. However, this can be customized by exposing a `PasswordEncoder` as a Spring bean.

If you are migrating from Spring Security 4.2.x you can revert to the previous behavior by exposing a `NoOpPasswordEncoder` bean.

Reverting to `NoOpPasswordEncoder` is not considered to be secure. You should instead migrate to using `DelegatingPasswordEncoder` to support secure password encoding.

Example 30. NoOpPasswordEncoder

Java

```
@Bean
public static NoOpPasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
}
```

XML

```
<b:bean id="passwordEncoder"
        class="org.springframework.security.crypto.password.NoOpPasswordEncoder"
factory-method="getInstance"/>
```

Kotlin

```
@Bean
fun passwordEncoder(): PasswordEncoder {
    return NoOpPasswordEncoder.getInstance();
}
```

XML Configuration requires the `NoOpPasswordEncoder` bean name to be `passwordEncoder`.

Change Password Configuration

Most applications that allow a user to specify a password also require a feature for updating that password.

A Well-Know URL for Changing Passwords indicates a mechanism by which password managers can discover the password update endpoint for a given application.

You can configure Spring Security to provide this discovery endpoint. For example, if the change

password endpoint in your application is `/change-password`, then you can configure Spring Security like so:

Example 31. Default Change Password Endpoint

Java

```
http
    .passwordManagement(Customizer.withDefaults())
```

XML

```
<sec:password-management/>
```

Kotlin

```
http {
    passwordManagement { }
}
```

Then, when a password manager navigates to `/.well-known/change-password` then Spring Security will redirect your endpoint, `/change-password`.

Or, if your endpoint is something other than `/change-password`, you can also specify that like so:

Example 32. Change Password Endpoint

Java

```
http
    .passwordManagement((management) -> management
        .changePasswordPage("/update-password")
    )
```

XML

```
<sec:password-management change-password-page="/update-password"/>
```

Kotlin

```
http {
    passwordManagement {
        changePasswordPage = "/update-password"
    }
}
```

With the above configuration, when a password manager navigates to `/.well-known/change-password`, then Spring Security will redirect to `/update-password`.

5.2. Protection Against Exploits

Spring Security provides protection against common exploits. Whenever possible, the protection is enabled by default. Below you will find high level description of the various exploits that Spring Security protects against.

5.2.1. Cross Site Request Forgery (CSRF)

Spring provides comprehensive support for protecting against Cross Site Request Forgery (CSRF) attacks. In the following sections we will explore:

- What is a CSRF Attack?
- Protecting Against CSRF Attacks
- CSRF Considerations

This portion of the documentation discusses the general topic of CSRF protection. Refer to the relevant sections for specific information on CSRF protection for servlet and WebFlux based applications.

What is a CSRF Attack?

The best way to understand a CSRF attack is by taking a look at a concrete example.

Assume that your bank's website provides a form that allows transferring money from the currently logged in user to another bank account. For example, the transfer form might look like:

Example 33. Transfer form

```
<form method="post"
    action="/transfer">
<input type="text"
    name="amount"/>
<input type="text"
    name="routingNumber"/>
<input type="text"
    name="account"/>
<input type="submit"
    value="Transfer"/>
</form>
```

The corresponding HTTP request might look like:

Example 34. Transfer HTTP request

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876
```

Now pretend you authenticate to your bank's website and then, without logging out, visit an evil website. The evil website contains an HTML page with the following form:

Example 35. Evil transfer form

```
<form method="post"
    action="https://bank.example.com/transfer">
<input type="hidden"
    name="amount"
    value="100.00"/>
<input type="hidden"
    name="routingNumber"
    value="evilsRoutingNumber"/>
<input type="hidden"
    name="account"
    value="evilsAccountNumber"/>
<input type="submit"
    value="Win Money!"/>
</form>
```

You like to win money, so you click on the submit button. In the process, you have unintentionally transferred $100 to a malicious user. This happens because, while the evil website cannot see your cookies, the cookies associated with your bank are still sent along with the request.

Worst yet, this whole process could have been automated using JavaScript. This means you didn't even need to click on the button. Furthermore, it could just as easily happen when visiting an honest site that is a victim of a XSS attack. So how do we protect our users from such attacks?

Protecting Against CSRF Attacks

The reason that a CSRF attack is possible is that the HTTP request from the victim's website and the request from the attacker's website are exactly the same. This means there is no way to reject requests coming from the evil website and allow requests coming from the bank's website. To protect against CSRF attacks we need to ensure there is something in the request that the evil site is unable to provide so we can differentiate the two requests.

Spring provides two mechanisms to protect against CSRF attacks:

- The Synchronizer Token Pattern
- Specifying the SameSite Attribute on your session cookie

Both protections require that Safe Methods Must be Idempotent

Safe Methods Must be Idempotent

In order for either protection against CSRF to work, the application must ensure that "safe" HTTP methods are idempotent. This means that requests with the HTTP method `GET`, `HEAD`, `OPTIONS`, and `TRACE` should not change the state of the application.

Synchronizer Token Pattern

The predominant and most comprehensive way to protect against CSRF attacks is to use the Synchronizer Token Pattern. This solution is to ensure that each HTTP request requires, in addition to our session cookie, a secure random generated value called a CSRF token must be present in the HTTP request.

When an HTTP request is submitted, the server must look up the expected CSRF token and compare it against the actual CSRF token in the HTTP request. If the values do not match, the HTTP request should be rejected.

The key to this working is that the actual CSRF token should be in a part of the HTTP request that is not automatically included by the browser. For example, requiring the actual CSRF token in an HTTP parameter or an HTTP header will protect against CSRF attacks. Requiring the actual CSRF token in a cookie does not work because cookies are automatically included in the HTTP request by the browser.

We can relax the expectations to only require the actual CSRF token for each HTTP request that updates state of the application. For that to work, our application must ensure that safe HTTP methods are idempotent. This improves usability since we want to allow linking to our website using links from external sites. Additionally, we do not want to include the random token in HTTP GET as this can cause the tokens to be leaked.

Let's take a look at how our example would change when using the Synchronizer Token Pattern. Assume the actual CSRF token is required to be in an HTTP parameter named `_csrf`. Our application's transfer form would look like:

Example 36. Synchronizer Token Form

```
<form method="post"
    action="/transfer">
<input type="hidden"
    name="_csrf"
    value="4bfd1575-3ad1-4d21-96c7-4ef2d9f86721"/>
<input type="text"
    name="amount"/>
<input type="text"
    name="routingNumber"/>
<input type="hidden"
    name="account"/>
<input type="submit"
    value="Transfer"/>
</form>
```

The form now contains a hidden input with the value of the CSRF token. External sites cannot read the CSRF token since the same origin policy ensures the evil site cannot read the response.

The corresponding HTTP request to transfer money would look like this:

Example 37. Synchronizer Token request

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876&_csrf=4bfd1575-3ad1-4d21-96c7-
4ef2d9f86721
```

You will notice that the HTTP request now contains the `_csrf` parameter with a secure random value. The evil website will not be able to provide the correct value for the `_csrf` parameter (which must be explicitly provided on the evil website) and the transfer will fail when the server compares the actual CSRF token to the expected CSRF token.

SameSite Attribute

An emerging way to protect against CSRF Attacks is to specify the SameSite Attribute on cookies. A server can specify the `SameSite` attribute when setting a cookie to indicate that the cookie should not be sent when coming from external sites.

Spring Security does not directly control the creation of the session cookie, so it does not provide support for the SameSite attribute. Spring Session provides support for the `SameSite` attribute in servlet based applications. Spring Framework's CookieWebSessionIdResolver provides out of the box support for the `SameSite` attribute in WebFlux based applications.

An example, HTTP response header with the `SameSite` attribute might look like:

Example 38. SameSite HTTP response

```
Set-Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly;
SameSite=Lax
```

Valid values for the `SameSite` attribute are:

- `Strict` - when specified any request coming from the same-site will include the cookie. Otherwise, the cookie will not be included in the HTTP request.
- `Lax` - when specified cookies will be sent when coming from the same-site or when the request comes from top-level navigations and the method is idempotent. Otherwise, the cookie will not be included in the HTTP request.

Let's take a look at how our example could be protected using the `SameSite` attribute. The bank application can protect against CSRF by specifying the `SameSite` attribute on the session cookie.

With the `SameSite` attribute set on our session cookie, the browser will continue to send the `JSESSIONID` cookie with requests coming from the banking website. However, the browser will no longer send the `JSESSIONID` cookie with a transfer request coming from the evil website. Since the session is no longer present in the transfer request coming from the evil website, the application is protected from the CSRF attack.

There are some important considerations that one should be aware about when using `SameSite` attribute to protect against CSRF attacks.

Setting the `SameSite` attribute to `Strict` provides a stronger defense but can confuse users. Consider a user that stays logged into a social media site hosted at https://social.example.com. The user receives an email at https://email.example.org that includes a link to the social media site. If the user clicks on the link, they would rightfully expect to be authenticated to the social media site. However, if the `SameSite` attribute is `Strict` the cookie would not be sent and so the user would not be authenticated.

We could improve the protection and usability of `SameSite` protection against CSRF attacks by implementing gh-7537.

Another obvious consideration is that in order for the `SameSite` attribute to protect users, the browser must support the `SameSite` attribute. Most modern browsers do support the SameSite attribute. However, older browsers that are still in use may not.

For this reason, it is generally recommended to use the `SameSite` attribute as a defense in depth rather than the sole protection against CSRF attacks.

When to use CSRF protection

When should you use CSRF protection? Our recommendation is to use CSRF protection for any request that could be processed by a browser by normal users. If you are only creating a service that is used by non-browser clients, you will likely want to disable CSRF protection.

CSRF protection and JSON

A common question is "do I need to protect JSON requests made by javascript?" The short answer is, it depends. However, you must be very careful as there are CSRF exploits that can impact JSON requests. For example, a malicious user can create a CSRF with JSON using the following form:

Example 39. CSRF with JSON form

```
<form action="https://bank.example.com/transfer" method="post"
enctype="text/plain">
    <input
name='{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsAccountNu
mber", "ignore_me":"' value='test"}' type='hidden'>
    <input type="submit"
        value="Win Money!"/>
</form>
```

This will produce the following JSON structure

Example 40. CSRF with JSON request

```
{ "amount": 100,
"routingNumber": "evilsRoutingNumber",
"account": "evilsAccountNumber",
"ignore_me": "=test"
}
```

If an application were not validating the Content-Type, then it would be exposed to this exploit. Depending on the setup, a Spring MVC application that validates the Content-Type could still be exploited by updating the URL suffix to end with `.json` as shown below:

Example 41. CSRF with JSON Spring MVC form

```
<form action="https://bank.example.com/transfer.json" method="post"
enctype="text/plain">
    <input
name='{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsAccountNu
mber", "ignore_me":"' value='test"}' type='hidden'>
    <input type="submit"
        value="Win Money!"/>
</form>
```

CSRF and Stateless Browser Applications

What if my application is stateless? That doesn't necessarily mean you are protected. In fact, if a user does not need to perform any actions in the web browser for a given request, they are likely still vulnerable to CSRF attacks.

For example, consider an application that uses a custom cookie that contains all the state within it for authentication instead of the JSESSIONID. When the CSRF attack is made the custom cookie will be sent with the request in the same manner that the JSESSIONID cookie was sent in our previous example. This application will be vulnerable to CSRF attacks.

Applications that use basic authentication are also vulnerable to CSRF attacks. The application is vulnerable since the browser will automatically include the username and password in any requests in the same manner that the JSESSIONID cookie was sent in our previous example.

CSRF Considerations

There are a few special considerations to consider when implementing protection against CSRF attacks.

Logging In

In order to protect against forging log in requests the log in HTTP request should be protected against CSRF attacks. Protecting against forging log in requests is necessary so that a malicious user cannot read a victim's sensitive information. The attack is performed as follows:

- A malicious user performs a CSRF log in using the malicious user's credentials. The victim is now authenticated as the malicious user.
- The malicious user then tricks the victim to visit the compromised website and enter sensitive information
- The information is associated to the malicious user's account so the malicious user can log in with their own credentials and view the vicitim's sensitive information

A possible complication to ensuring log in HTTP requests are protected against CSRF attacks is that the user might experience a session timeout that causes the request to be rejected. A session timeout is surprising to users who do not expect to need to have a session in order to log in. For

more information refer to CSRF and Session Timeouts.

Logging Out

In order to protect against forging log out requests, the log out HTTP request should be protected against CSRF attacks. Protecting against forging log out requests is necessary so a malicious user cannot read a victim's sensitive information. For details on the attack refer to this blog post.

A possible complication to ensuring log out HTTP requests are protected against CSRF attacks is that the user might experience a session timeout that causes the request to be rejected. A session timeout is surprising to users who do not expect to need to have a session in order to log out. For more information refer to CSRF and Session Timeouts.

CSRF and Session Timeouts

More often than not, the expected CSRF token is stored in the session. This means that as soon as the session expires the server will not find an expected CSRF token and reject the HTTP request. There are a number of options to solve timeouts each of which come with trade offs.

- The best way to mitigate the timeout is by using JavaScript to request a CSRF token on form submission. The form is then updated with the CSRF token and submitted.
- Another option is to have some JavaScript that lets the user know their session is about to expire. The user can click a button to continue and refresh the session.
- Finally, the expected CSRF token could be stored in a cookie. This allows the expected CSRF token to outlive the session.

  One might ask why the expected CSRF token isn't stored in a cookie by default. This is because there are known exploits in which headers (for example, to specify the cookies) can be set by another domain. This is the same reason Ruby on Rails no longer skips CSRF checks when the header X-Requested-With is present. See this webappsec.org thread for details on how to perform the exploit. Another disadvantage is that by removing the state (that is, the timeout), you lose the ability to forcibly invalidate the token if it is compromised.

Multipart (file upload)

Protecting multipart requests (file uploads) from CSRF attacks causes a chicken and the egg problem. In order to prevent a CSRF attack from occurring, the body of the HTTP request must be read to obtain actual CSRF token. However, reading the body means that the file will be uploaded which means an external site can upload a file.

There are two options to using CSRF protection with multipart/form-data. Each option has its trade-offs.

- Place CSRF Token in the Body
- Place CSRF Token in the URL

Before you integrate Spring Security's CSRF protection with multipart file upload, ensure that you can upload without the CSRF protection first. More information about using multipart forms with Spring can be found within the 1.1.11. Multipart Resolver section of the Spring reference and the MultipartFilter javadoc.

Place CSRF Token in the Body

The first option is to include the actual CSRF token in the body of the request. By placing the CSRF token in the body, the body will be read before authorization is performed. This means that anyone can place temporary files on your server. However, only authorized users will be able to submit a File that is processed by your application. In general, this is the recommended approach because the temporary file upload should have a negligible impact on most servers.

Include CSRF Token in URL

If allowing unauthorized users to upload temporary files is not acceptable, an alternative is to include the expected CSRF token as a query parameter in the action attribute of the form. The disadvantage to this approach is that query parameters can be leaked. More generally, it is considered best practice to place sensitive data within the body or headers to ensure it is not leaked. Additional information can be found in RFC 2616 Section 15.1.3 Encoding Sensitive Information in URI's.

HiddenHttpMethodFilter

In some applications a form parameter can be used to override the HTTP method. For example, the form below could be used to treat the HTTP method as a `delete` rather than a `post`.

Example 42. CSRF Hidden HTTP Method Form

```
<form action="/process"
    method="post">
    <!-- ... -->
    <input type="hidden"
        name="_method"
        value="delete"/>
</form>
```

Overriding the HTTP method occurs in a filter. That filter must be placed before Spring Security's support. Note that overriding only happens on a `post`, so this is actually unlikely to cause any real problems. However, it is still best practice to ensure it is placed before Spring Security's filters.

5.2.2. Security HTTP Response Headers

This portion of the documentation discusses the general topic of Security HTTP Response Headers. Refer to the relevant sections for specific information on Security HTTP Response Headers servlet and WebFlux based applications.

There are many HTTP response headers that can be used to increase the security of web applications. This section is dedicated to the various HTTP response headers that Spring Security provides explicit support for. If necessary, Spring Security can also be configured to provide custom headers.

Default Security Headers

Refer to the relevant sections to see how to customize the defaults for both servlet and webflux based applications.

Spring Security provides a default set of security related HTTP response headers to provide secure defaults.

The default for Spring Security is to include the following headers:

Example 43. Default Security HTTP Response Headers

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
```

Strict-Transport-Security is only added on HTTPS requests

If the defaults do not meet your needs, you can easily remove, modify, or add headers from these defaults. For additional details on each of these headers, refer to the corresponding sections:

- Cache Control
- Content Type Options
- HTTP Strict Transport Security
- X-Frame-Options
- X-XSS-Protection

Cache Control

Refer to the relevant sections to see how to customize the defaults for both servlet and webflux based applications.

Spring Security's default is to disable caching to protect user's content.

If a user authenticates to view sensitive information and then logs out, we don't want a malicious user to be able to click the back button to view the sensitive information. The cache control headers that are sent by default are:

Example 44. Default Cache Control HTTP Response Headers

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
```

In order to be secure by default, Spring Security adds these headers by default. However, if your application provides its own cache control headers Spring Security will back out of the way. This allows for applications to ensure that static resources like CSS and JavaScript can be cached.

Content Type Options

Refer to the relevant sections to see how to customize the defaults for both servlet and webflux based applications.

Historically browsers, including Internet Explorer, would try to guess the content type of a request using content sniffing. This allowed browsers to improve the user experience by guessing the content type on resources that had not specified the content type. For example, if a browser encountered a JavaScript file that did not have the content type specified, it would be able to guess the content type and then run it.

There are many additional things one should do (i.e. only display the document in a distinct domain, ensure Content-Type header is set, sanitize the document, etc) when allowing content to be uploaded. However, these measures are out of the scope of what Spring Security provides. It is also important to point out when disabling content sniffing, you must specify the content type in order for things to work properly.

The problem with content sniffing is that this allowed malicious users to use polyglots (i.e. a file that is valid as multiple content types) to perform XSS attacks. For example, some sites may allow users to submit a valid postscript document to a website and view it. A malicious user might create a postscript document that is also a valid JavaScript file and perform a XSS attack with it.

Spring Security disables content sniffing by default by adding the following header to HTTP responses:

Example 45. nosniff HTTP Response Header

```
X-Content-Type-Options: nosniff
```

HTTP Strict Transport Security (HSTS)

Refer to the relevant sections to see how to customize the defaults for both servlet and webflux based applications.

When you type in your bank's website, do you enter mybank.example.com or do you enter https://mybank.example.com? If you omit the https protocol, you are potentially vulnerable to Man in the Middle attacks. Even if the website performs a redirect to https://mybank.example.com a malicious user could intercept the initial HTTP request and manipulate the response (e.g. redirect to https://mibank.example.com and steal their credentials).

Many users omit the https protocol and this is why HTTP Strict Transport Security (HSTS) was created. Once mybank.example.com is added as a HSTS host, a browser can know ahead of time that any request to mybank.example.com should be interpreted as https://mybank.example.com. This greatly reduces the possibility of a Man in the Middle attack occurring.

In accordance with RFC6797, the HSTS header is only injected into HTTPS responses. In order for the browser to acknowledge the header, the browser must first trust the CA that signed the SSL certificate used to make the connection (not just the SSL certificate).

One way for a site to be marked as a HSTS host is to have the host preloaded into the browser. Another is to add the `Strict-Transport-Security` header to the response. For example, Spring Security's default behavior is to add the following header which instructs the browser to treat the domain as an HSTS host for a year (there are approximately 31536000 seconds in a year):

Example 46. Strict Transport Security HTTP Response Header

```
Strict-Transport-Security: max-age=31536000 ; includeSubDomains ; preload
```

The optional `includeSubDomains` directive instructs the browser that subdomains (e.g. secure.mybank.example.com) should also be treated as an HSTS domain.

The optional `preload` directive instructs the browser that domain should be preloaded in browser as HSTS domain. For more details on HSTS preload please see https://hstspreload.org.

HTTP Public Key Pinning (HPKP)

In order to remain passive Spring Security still provides support for HPKP in servlet environments, but for the reasons listed above HPKP is no longer recommended by the security team.

HTTP Public Key Pinning (HPKP) specifies to a web client which public key to use with certain web server to prevent Man in the Middle (MITM) attacks with forged certificates. When used correctly, HPKP could add additional layers of protection against compromised certificates. However, due to the complexity of HPKP many experts no longer recommend using it and Chrome has even removed support for it.

For additional details around why HPKP is no longer recommended read Is HTTP Public Key Pinning Dead? and I'm giving up on HPKP.

X-Frame-Options

Refer to the relevant sections to see how to customize the defaults for both servlet and webflux based applications.

Allowing your website to be added to a frame can be a security issue. For example, using clever CSS styling users could be tricked into clicking on something that they were not intending. For example, a user that is logged into their bank might click a button that grants access to other users. This sort of attack is known as Clickjacking.

Another modern approach to dealing with clickjacking is to use Content Security Policy (CSP).

There are a number ways to mitigate clickjacking attacks. For example, to protect legacy browsers from clickjacking attacks you can use frame breaking code. While not perfect, the frame breaking code is the best you can do for the legacy browsers.

A more modern approach to address clickjacking is to use X-Frame-Options header. By default Spring Security disables rendering pages within an iframe using with the following header:

```
X-Frame-Options: DENY
```

X-XSS-Protection

Refer to the relevant sections to see how to customize the defaults for both servlet and webflux based applications.

Some browsers have built in support for filtering out reflected XSS attacks. This is by no means foolproof, but does assist in XSS protection.

The filtering is typically enabled by default, so adding the header typically just ensures it is enabled and instructs the browser what to do when a XSS attack is detected. For example, the filter might try to change the content in the least invasive way to still render everything. At times, this type of replacement can become a XSS vulnerability in itself. Instead, it is best to block the content rather than attempt to fix it. By default Spring Security blocks the content using the following header:

```
X-XSS-Protection: 1; mode=block
```

Content Security Policy (CSP)

Refer to the relevant sections to see how to configure both servlet and webflux based applications.

Content Security Policy (CSP) is a mechanism that web applications can leverage to mitigate content injection vulnerabilities, such as cross-site scripting (XSS). CSP is a declarative policy that provides a facility for web application authors to declare and ultimately inform the client (user-agent) about

the sources from which the web application expects to load resources.

Content Security Policy is not intended to solve all content injection vulnerabilities. Instead, CSP can be leveraged to help reduce the harm caused by content injection attacks. As a first line of defense, web application authors should validate their input and encode their output.

A web application may employ the use of CSP by including one of the following HTTP headers in the response:

- `Content-Security-Policy`
- `Content-Security-Policy-Report-Only`

Each of these headers are used as a mechanism to deliver a security policy to the client. A security policy contains a set of security policy directives, each responsible for declaring the restrictions for a particular resource representation.

For example, a web application can declare that it expects to load scripts from specific, trusted sources, by including the following header in the response:

Example 47. Content Security Policy Example

```
Content-Security-Policy: script-src https://trustedscripts.example.com
```

An attempt to load a script from another source other than what is declared in the `script-src` directive will be blocked by the user-agent. Additionally, if the report-uri directive is declared in the security policy, then the violation will be reported by the user-agent to the declared URL.

For example, if a web application violates the declared security policy, the following response header will instruct the user-agent to send violation reports to the URL specified in the policy's `report-uri` directive.

Example 48. Content Security Policy with report-uri

```
Content-Security-Policy: script-src https://trustedscripts.example.com; report-uri
/csp-report-endpoint/
```

Violation reports are standard JSON structures that can be captured either by the web application's own API or by a publicly hosted CSP violation reporting service, such as, https://report-uri.com/.

The `Content-Security-Policy-Report-Only` header provides the capability for web application authors and administrators to monitor security policies, rather than enforce them. This header is typically used when experimenting and/or developing security policies for a site. When a policy is deemed effective, it can be enforced by using the `Content-Security-Policy` header field instead.

Given the following response header, the policy declares that scripts may be loaded from one of

two possible sources.

Example 49. Content Security Policy Report Only

```
Content-Security-Policy-Report-Only: script-src 'self'
https://trustedscripts.example.com; report-uri /csp-report-endpoint/
```

If the site violates this policy, by attempting to load a script from *evil.com*, the user-agent will send a violation report to the declared URL specified by the *report-uri* directive, but still allow the violating resource to load nevertheless.

Applying Content Security Policy to a web application is often a non-trivial undertaking. The following resources may provide further assistance in developing effective security policies for your site.

An Introduction to Content Security Policy

CSP Guide - Mozilla Developer Network

W3C Candidate Recommendation

Referrer Policy

Refer to the relevant sections to see how to configure both servlet and webflux based applications.

Referrer Policy is a mechanism that web applications can leverage to manage the referrer field, which contains the last page the user was on.

Spring Security's approach is to use Referrer Policy header, which provides different policies:

Example 50. Referrer Policy Example

```
Referrer-Policy: same-origin
```

The Referrer-Policy response header instructs the browser to let the destination knows the source where the user was previously.

Feature Policy

Refer to the relevant sections to see how to configure both servlet and webflux based applications.

Feature Policy is a mechanism that allows web developers to selectively enable, disable, and modify the behavior of certain APIs and web features in the browser.

Example 51. Feature Policy Example

```
Feature-Policy: geolocation 'self'
```

With Feature Policy, developers can opt-in to a set of "policies" for the browser to enforce on specific features used throughout your site. These policies restrict what APIs the site can access or modify the browser's default behavior for certain features.

Permissions Policy

Refer to the relevant sections to see how to configure both servlet and webflux based applications.

Permissions Policy is a mechanism that allows web developers to selectively enable, disable, and modify the behavior of certain APIs and web features in the browser.

Example 52. Permissions Policy Example

```
Permissions-Policy: geolocation=(self)
```

With Permissions Policy, developers can opt-in to a set of "policies" for the browser to enforce on specific features used throughout your site. These policies restrict what APIs the site can access or modify the browser's default behavior for certain features.

Clear Site Data

Refer to the relevant sections to see how to configure both servlet and webflux based applications.

Clear Site Data is a mechanism by which any browser-side data - cookies, local storage, and the like - can be removed when an HTTP response contains this header:

```
Clear-Site-Data: "cache", "cookies", "storage", "executionContexts"
```

This is a nice clean-up action to perform on logout.

Custom Headers

Refer to the relevant sections to see how to configure both servlet based applications.

Spring Security has mechanisms to make it convenient to add the more common security headers to your application. However, it also provides hooks to enable adding custom headers.

5.2.3. HTTP

All HTTP based communication, including static resources, should be protected using TLS.

As a framework, Spring Security does not handle HTTP connections and thus does not provide support for HTTPS directly. However, it does provide a number of features that help with HTTPS usage.

Redirect to HTTPS

When a client uses HTTP, Spring Security can be configured to redirect to HTTPS both Servlet and WebFlux environments.

Strict Transport Security

Spring Security provides support for Strict Transport Security and enables it by default.

Proxy Server Configuration

When using a proxy server it is important to ensure that you have configured your application properly. For example, many applications will have a load balancer that responds to request for https://example.com/ by forwarding the request to an application server at https://192.168.1:8080 Without proper configuration, the application server will not know that the load balancer exists and treat the request as though https://192.168.1:8080 was requested by the client.

To fix this you can use RFC 7239 to specify that a load balancer is being used. To make the application aware of this, you need to either configure your application server aware of the X-Forwarded headers. For example Tomcat uses the RemoteIpValve and Jetty uses ForwardedRequestCustomizer. Alternatively, Spring users can leverage ForwardedHeaderFilter.

Spring Boot users may use the `server.use-forward-headers` property to configure the application. See the Spring Boot documentation for further details.

Chapter 6. Project Modules and Dependencies

In Spring Security 3.0, the codebase was sub-divided into separate jars which more clearly separate different functionality areas and third-party dependencies. If you use Maven to build your project, these are the modules you should add to your `pom.xml`. Even if you do not use Maven, we recommend that you consult the `pom.xml` files to get an idea of third-party dependencies and versions. Another good idea is to examine the libraries that are included in the sample applications.

This section provides a reference of the modules in Spring Security and the additional dependencies that they require in order to function in a running application. We don't include dependencies that are only used when building or testing Spring Security itself. Nor do we include transitive dependencies which are required by external dependencies.

The version of Spring required is listed on the project website, so the specific versions are omitted for Spring dependencies below. Note that some of the dependencies listed as "optional" below may still be required for other non-security functionality in a Spring application. Also dependencies listed as "optional" may not actually be marked as such in the project's Maven POM files if they are used in most applications. They are "optional" only in the sense that you don't need them unless you are using the specified functionality.

Where a module depends on another Spring Security module, the non-optional dependencies of the module it depends on are also assumed to be required and are not listed separately.

6.1. Core — `spring-security-core.jar`

This module contains core authentication and access-contol classes and interfaces, remoting support, and basic provisioning APIs. It is required by any application that uses Spring Security. It supports standalone applications, remote clients, method (service layer) security, and JDBC user provisioning. It contains the following top-level packages:

- `org.springframework.security.core`
- `org.springframework.security.access`
- `org.springframework.security.authentication`
- `org.springframework.security.provisioning`

Table 1. Core Dependencies

Dependency	Version	Description
ehcache	1.6.2	Required if the Ehcache-based user cache implementation is used (optional).
spring-aop		Method security is based on Spring AOP

Dependency	Version	Description
spring-beans		Required for Spring configuration
spring-expression		Required for expression-based method security (optional)
spring-jdbc		Required if using a database to store user data (optional).
spring-tx		Required if using a database to store user data (optional).
aspectjrt	1.6.10	Required if using AspectJ support (optional).
jsr250-api	1.0	Required if you are using JSR-250 method-security annotations (optional).

6.2. Remoting — `spring-security-remoting.jar`

This module provides integration with Spring Remoting. You do not need this unless you are writing a remote client that uses Spring Remoting. The main package is `org.springframework.security.remoting`.

Table 2. Remoting Dependencies

Dependency	Version	Description
spring-security-core		
spring-web		Required for clients which use HTTP remoting support.

6.3. Web — `spring-security-web.jar`

This module contains filters and related web-security infrastructure code. It contains anything with a servlet API dependency. You need it if you require Spring Security web authentication services and URL-based access-control. The main package is `org.springframework.security.web`.

Table 3. Web Dependencies

Dependency	Version	Description
spring-security-core		
spring-web		Spring web support classes are used extensively.
spring-jdbc		Required for JDBC-based persistent remember-me token repository (optional).

Dependency	Version	Description
spring-tx		Required by remember-me persistent token repository implementations (optional).

6.4. Config — `spring-security-config.jar`

This module contains the security namespace parsing code and Java configuration code. You need it if you use the Spring Security XML namespace for configuration or Spring Security's Java Configuration support. The main package is `org.springframework.security.config`. None of the classes are intended for direct use in an application.

Table 4. Config Dependencies

Dependency	Version	Description
spring-security-core		
spring-security-web		Required if you are using any web-related namespace configuration (optional).
spring-security-ldap		Required if you are using the LDAP namespace options (optional).
spring-security-openid		Required if you are using OpenID authentication (optional).
aspectjweaver	1.6.10	Required if using the protect-pointcut namespace syntax (optional).

6.5. LDAP — `spring-security-ldap.jar`

This module provides LDAP authentication and provisioning code. It is required if you need to use LDAP authentication or manage LDAP user entries. The top-level package is `org.springframework.security.ldap`.

Table 5. LDAP Dependencies

Dependency	Version	Description
spring-security-core		
spring-ldap-core	1.3.0	LDAP support is based on Spring LDAP.
spring-tx		Data exception classes are required.

Dependency	Version	Description
apache-ds [1]	1.5.5	Required if you are using an embedded LDAP server (optional).
shared-ldap	0.9.15	Required if you are using an embedded LDAP server (optional).
ldapsdk	4.1	Mozilla LdapSDK. Used for decoding LDAP password policy controls if you are using password-policy functionality with OpenLDAP, for example.

6.6. OAuth 2.0 Core — `spring-security-oauth2-core.jar`

`spring-security-oauth2-core.jar` contains core classes and interfaces that provide support for the OAuth 2.0 Authorization Framework and for OpenID Connect Core 1.0. It is required by applications that use OAuth 2.0 or OpenID Connect Core 1.0, such as client, resource server, and authorization server. The top-level package is `org.springframework.security.oauth2.core`.

6.7. OAuth 2.0 Client — `spring-security-oauth2-client.jar`

`spring-security-oauth2-client.jar` contains Spring Security's client support for OAuth 2.0 Authorization Framework and OpenID Connect Core 1.0. It is required by applications that use OAuth 2.0 Login or OAuth Client support. The top-level package is `org.springframework.security.oauth2.client`.

6.8. OAuth 2.0 JOSE — `spring-security-oauth2-jose.jar`

`spring-security-oauth2-jose.jar` contains Spring Security's support for the JOSE (Javascript Object Signing and Encryption) framework. The JOSE framework is intended to provide a method to securely transfer claims between parties. It is built from a collection of specifications:

- JSON Web Token (JWT)
- JSON Web Signature (JWS)
- JSON Web Encryption (JWE)
- JSON Web Key (JWK)

It contains the following top-level packages:

- `org.springframework.security.oauth2.jwt`
- `org.springframework.security.oauth2.jose`

6.9. OAuth 2.0 Resource Server — spring-security-oauth2-resource-server.jar

spring-security-oauth2-resource-server.jar contains Spring Security's support for OAuth 2.0 Resource Servers. It is used to protect APIs via OAuth 2.0 Bearer Tokens. The top-level package is org.springframework.security.oauth2.server.resource.

6.10. ACL — spring-security-acl.jar

This module contains a specialized domain object ACL implementation. It is used to apply security to specific domain object instances within your application. The top-level package is org.springframework.security.acls.

Table 6. ACL Dependencies

Dependency	Version	Description
spring-security-core		
ehcache	1.6.2	Required if the Ehcache-based ACL cache implementation is used (optional if you are using your own implementation).
spring-jdbc		Required if you are using the default JDBC-based AclService (optional if you implement your own).
spring-tx		Required if you are using the default JDBC-based AclService (optional if you implement your own).

6.11. CAS — spring-security-cas.jar

This module contains Spring Security's CAS client integration. You should use it if you want to use Spring Security web authentication with a CAS single sign-on server. The top-level package is org.springframework.security.cas.

Table 7. CAS Dependencies

Dependency	Version	Description
spring-security-core		
spring-security-web		
cas-client-core	3.1.12	The JA-SIG CAS Client. This is the basis of the Spring Security integration.

Dependency	Version	Description
ehcache	1.6.2	Required if you are using the Ehcache-based ticket cache (optional).

6.12. OpenID — `spring-security-openid.jar`

The OpenID 1.0 and 2.0 protocols have been deprecated and users are encouraged to migrate to OpenID Connect, which is supported by spring-security-oauth2.

This module contains OpenID web authentication support. It is used to authenticate users against an external OpenID server. The top-level package is `org.springframework.security.openid`. It requires OpenID4Java.

Table 8. OpenID Dependencies

Dependency	Version	Description
spring-security-core		
spring-security-web		
openid4java-nodeps	0.9.6	Spring Security's OpenID integration uses OpenID4Java.
httpclient	4.1.1	openid4java-nodeps depends on HttpClient 4.
guice	2.0	openid4java-nodeps depends on Guice 2.

6.13. Test — `spring-security-test.jar`

This module contains support for testing with Spring Security.

6.14. Taglibs — `spring-secuity-taglibs.jar`

Provides Spring Security's JSP tag implementations.

Table 9. Taglib Dependencies

Dependency	Version	Description
spring-security-core		
spring-security-web		
spring-security-acl		Required if you are using the `accesscontrollist` tag or `hasPermission()` expressions with ACLs (optional).

Dependency	Version	Description
spring-expression		Required if you are using SPEL expressions in your tag access constraints.

[1] The modules `apacheds-core`, `apacheds-core-entry`, `apacheds-protocol-shared`, `apacheds-protocol-ldap` and `apacheds-server-jndi` are required.

Chapter 7. Samples

Spring Security includes many samples applications.

These samples are being migrated to a separate project, however, you can still find the not migrated samples in an older branch of the Spring Security repository.

Servlet Applications

Spring Security integrates with the Servlet Container by using a standard Servlet `Filter`. This means it works with any application that runs in a Servlet Container. More concretely, you do not need to use Spring in your Servlet-based application to take advantage of Spring Security.

Chapter 8. Hello Spring Security

This section covers the minimum setup for how to use Spring Security with Spring Boot.

The completed application can be found in our samples repository. For your convenience, you can download a minimal Spring Boot + Spring Security application by clicking here.

8.1. Updating Dependencies

The only step you need to do is update the dependencies by using Maven or Gradle.

8.2. Starting Hello Spring Security Boot

You can now run the Spring Boot application by using the Maven Plugin's `run` goal. The following example shows how to do so (and the beginning of the output from doing so):

Example 53. Running Spring Boot Application

```
$ ./mvn spring-boot:run
...
INFO 23689 --- [  restartedMain] .s.s.UserDetailsServiceAutoConfiguration :

Using generated security password: 8e557245-73e2-4286-969a-ff57fe326336

...
```

8.3. Spring Boot Auto Configuration

Spring Boot automatically:

- Enables Spring Security's default configuration, which creates a servlet `Filter` as a bean named `springSecurityFilterChain`. This bean is responsible for all the security (protecting the application URLs, validating submitted username and passwords, redirecting to the log in form, and so on) within your application.
- Creates a `UserDetailsService` bean with a username of `user` and a randomly generated password that is logged to the console.
- Registers the `Filter` with a bean named `springSecurityFilterChain` with the Servlet container for every request.

Spring Boot is not configuring much, but it does a lot. A summary of the features follows:

- Require an authenticated user for any interaction with the application
- Generate a default login form for you

- Let the user with a username of `user` and a password that is logged to the console to authenticate with form-based authentication (in the preceding example, the password is `8e557245-73e2-4286-969a-ff57fe326336`)
- Protects the password storage with BCrypt
- Lets the user log out
- CSRF attack prevention
- Session Fixation protection
- Security Header integration
  - HTTP Strict Transport Security for secure requests
  - X-Content-Type-Options integration
  - Cache Control (can be overridden later by your application to allow caching of your static resources)
  - X-XSS-Protection integration
  - X-Frame-Options integration to help prevent Clickjacking
- Integrate with the following Servlet API methods:
  - `HttpServletRequest#getRemoteUser()`
  - `HttpServletRequest.html#getUserPrincipal()`
  - `HttpServletRequest.html#isUserInRole(java.lang.String)`
  - `HttpServletRequest.html#login(java.lang.String, java.lang.String)`
  - `HttpServletRequest.html#logout()`

Chapter 9. Servlet Security: The Big Picture

This section discusses Spring Security's high level architecture within Servlet based applications. We build on this high level understanding within Authentication, Authorization, Protection Against Exploits sections of the reference.

9.1. A Review of `Filter`s

Spring Security's Servlet support is based on Servlet `Filter`s, so it is helpful to look at the role of `Filter`s generally first. The picture below shows the typical layering of the handlers for a single HTTP request.

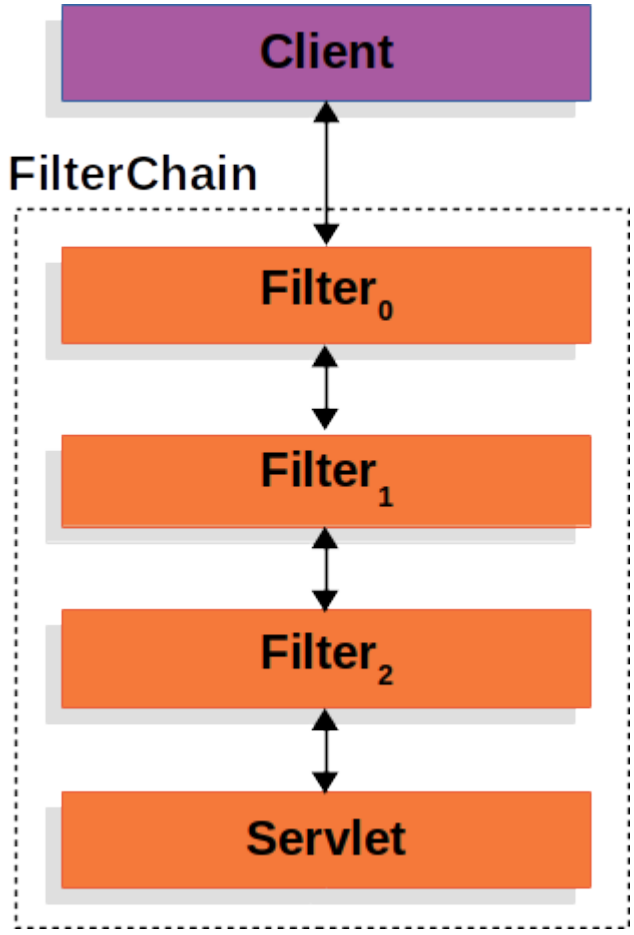


Figure 1. FilterChain

The client sends a request to the application, and the container creates a `FilterChain` which contains the `Filter`s and `Servlet` that should process the `HttpServletRequest` based on the path of the request URI. In a Spring MVC application the `Servlet` is an instance of `DispatcherServlet`. At most one `Servlet` can handle a single `HttpServletRequest` and `HttpServletResponse`. However, more than one `Filter` can be used to:

- Prevent downstream `Filter`s or the `Servlet` from being invoked. In this instance the `Filter` will typically write the `HttpServletResponse`.
- Modify the `HttpServletRequest` or `HttpServletResponse` used by the downstream `Filter`s and `Servlet`

The power of the `Filter` comes from the `FilterChain` that is passed into it.

Example 54. `FilterChain` Usage Example

Java

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) {
    // do something before the rest of the application
    chain.doFilter(request, response); // invoke the rest of the application
    // do something after the rest of the application
}
```

Kotlin

```
fun doFilter(request: ServletRequest, response: ServletResponse, chain:
FilterChain) {
    // do something before the rest of the application
    chain.doFilter(request, response) // invoke the rest of the application
    // do something after the rest of the application
}
```

Since a `Filter` only impacts downstream `Filter`s and the `Servlet`, the order each `Filter` is invoked is extremely important.

9.2. DelegatingFilterProxy

Spring provides a `Filter` implementation named `DelegatingFilterProxy` that allows bridging between the Servlet container's lifecycle and Spring's `ApplicationContext`. The Servlet container allows registering `Filter`s using its own standards, but it is not aware of Spring defined Beans. `DelegatingFilterProxy` can be registered via standard Servlet container mechanisms, but delegate all the work to a Spring Bean that implements `Filter`.

Here is a picture of how `DelegatingFilterProxy` fits into the `Filter`s and the `FilterChain`.

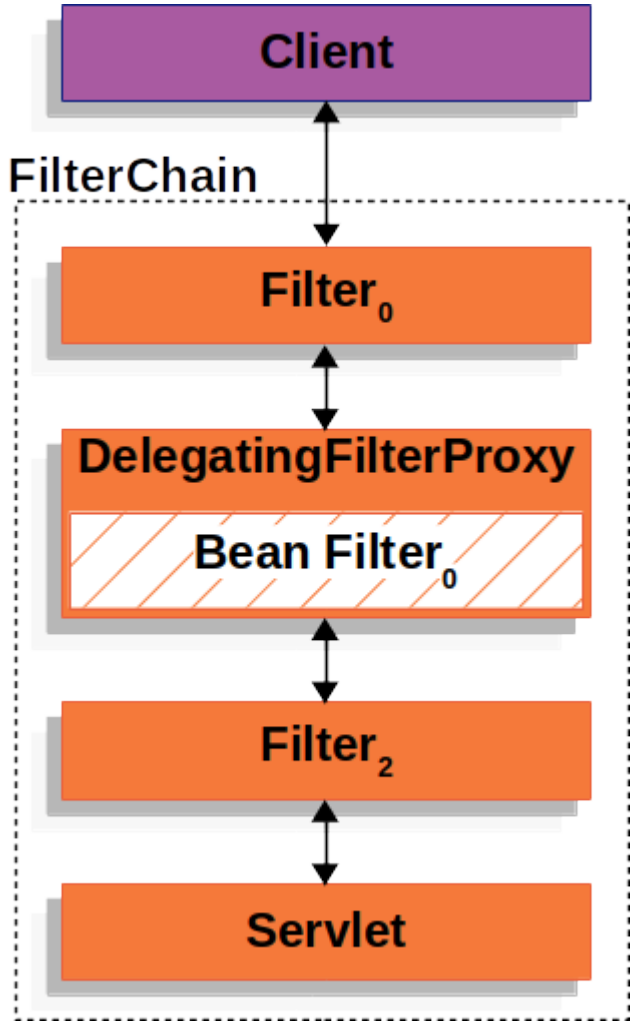


Figure 2. DelegatingFilterProxy

`DelegatingFilterProxy` looks up *Bean Filter*$_0$ from the `ApplicationContext` and then invokes *Bean Filter*$_0$. The pseudo code of `DelegatingFilterProxy` can be seen below.

Example 55. `DelegatingFilterProxy` Pseudo Code

Java

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) {
    // Lazily get Filter that was registered as a Spring Bean
    // For the example in DelegatingFilterProxy delegate is an instance of Bean
Filter0
    Filter delegate = getFilterBean(someBeanName);
    // delegate work to the Spring Bean
    delegate.doFilter(request, response);
}
```

Kotlin

```
fun doFilter(request: ServletRequest, response: ServletResponse, chain:
FilterChain) {
    // Lazily get Filter that was registered as a Spring Bean
    // For the example in DelegatingFilterProxy delegate is an instance of Bean
Filter0
    val delegate: Filter = getFilterBean(someBeanName)
    // delegate work to the Spring Bean
    delegate.doFilter(request, response)
}
```

Another benefit of `DelegatingFilterProxy` is that it allows delaying looking `Filter` bean instances. This is important because the container needs to register the `Filter` instances before the container can startup. However, Spring typically uses a `ContextLoaderListener` to load the Spring Beans which will not be done until after the `Filter` instances need to be registered.

9.3. FilterChainProxy

Spring Security's Servlet support is contained within `FilterChainProxy`. `FilterChainProxy` is a special `Filter` provided by Spring Security that allows delegating to many `Filter` instances through `SecurityFilterChain`. Since `FilterChainProxy` is a Bean, it is typically wrapped in a DelegatingFilterProxy.

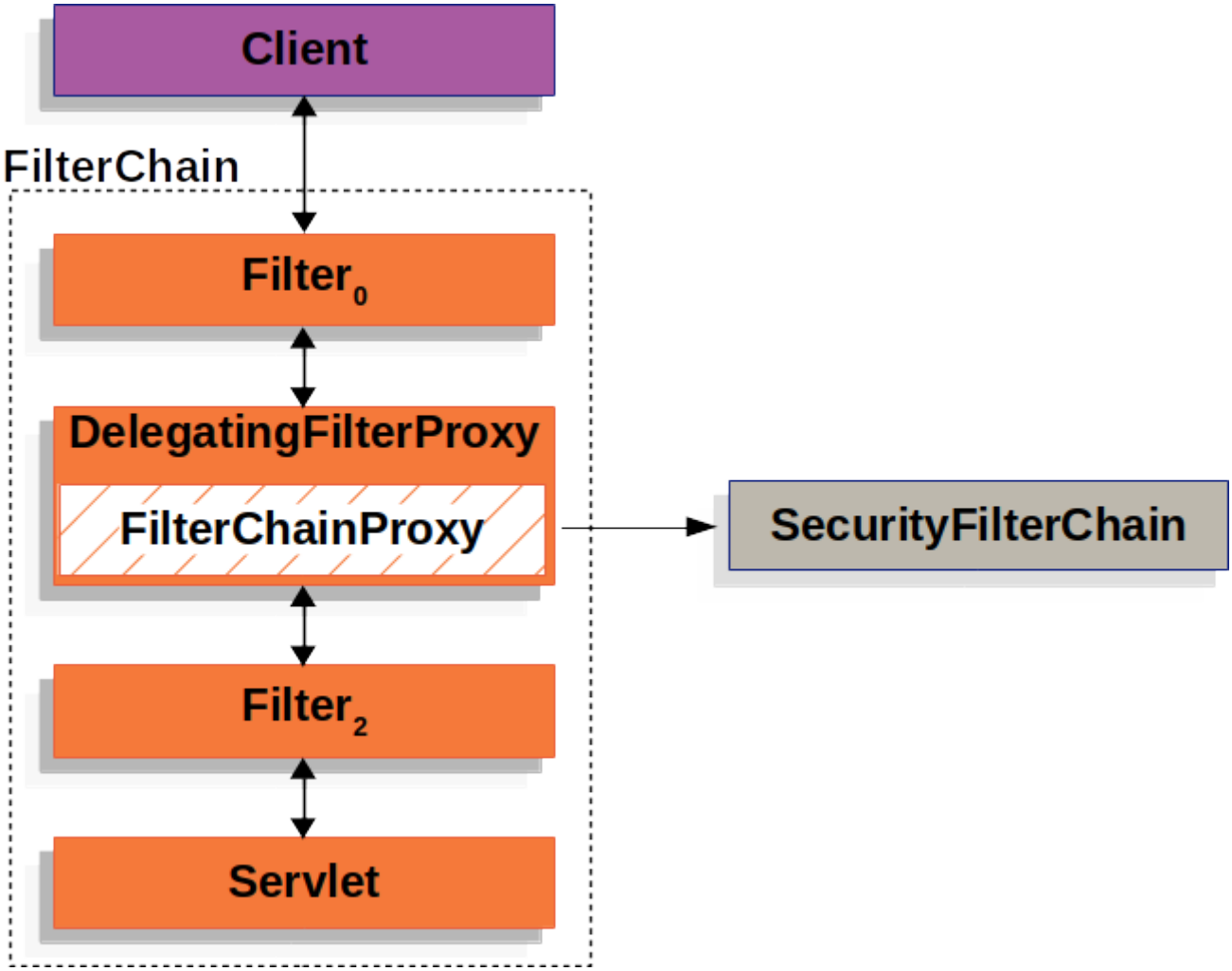


Figure 3. FilterChainProxy

9.4. SecurityFilterChain

`SecurityFilterChain` is used by FilterChainProxy to determine which Spring Security `Filter`s should be invoked for this request.

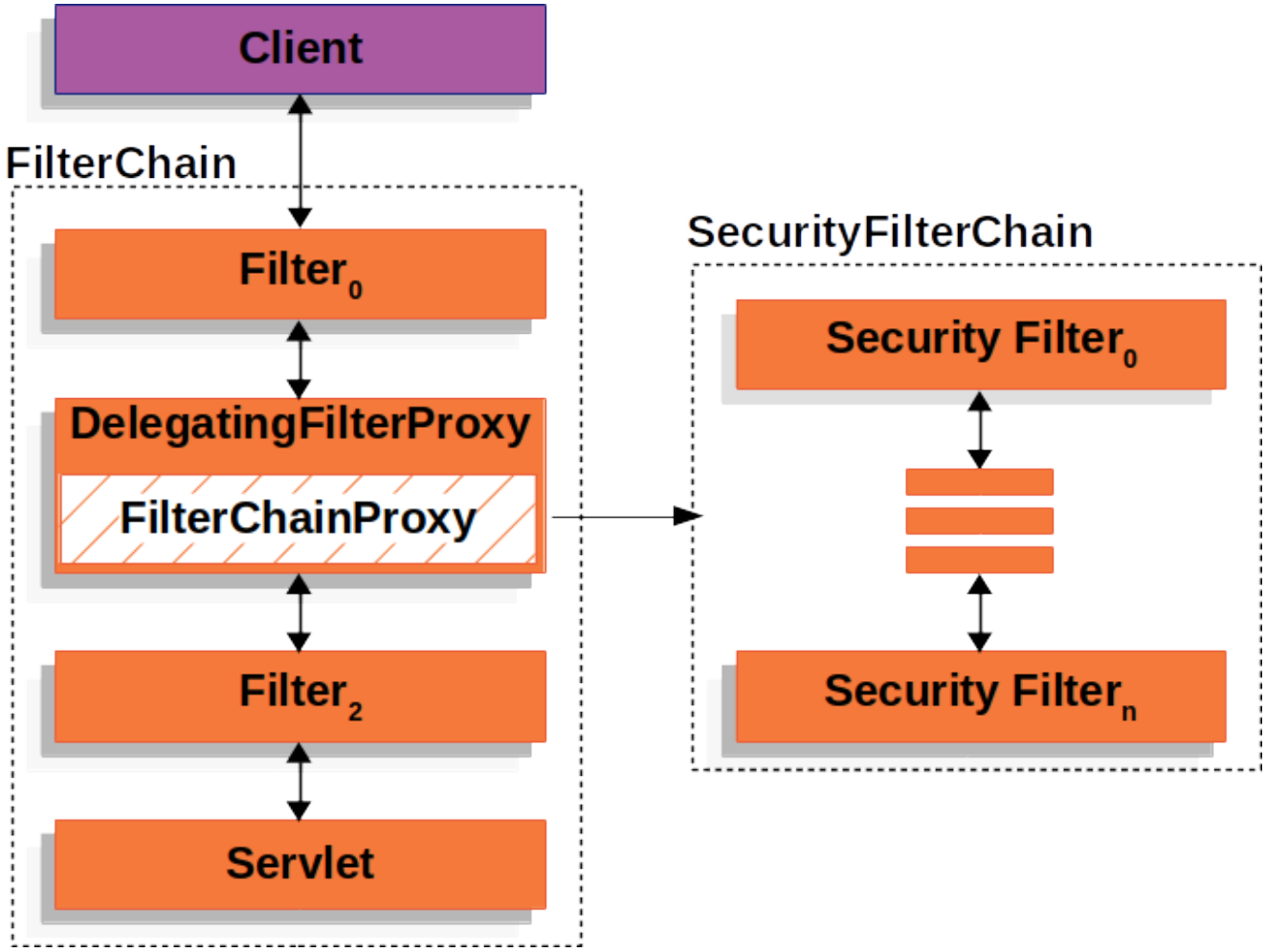


Figure 4. SecurityFilterChain

The Security Filters in `SecurityFilterChain` are typically Beans, but they are registered with `FilterChainProxy` instead of DelegatingFilterProxy. `FilterChainProxy` provides a number of advantages to registering directly with the Servlet container or DelegatingFilterProxy. First, it provides a starting point for all of Spring Security's Servlet support. For that reason, if you are attempting to troubleshoot Spring Security's Servlet support, adding a debug point in `FilterChainProxy` is a great place to start.

Second, since `FilterChainProxy` is central to Spring Security usage it can perform tasks that are not viewed as optional. For example, it clears out the `SecurityContext` to avoid memory leaks. It also applies Spring Security's `HttpFirewall` to protect applications against certain types of attacks.

In addition, it provides more flexibility in determining when a `SecurityFilterChain` should be invoked. In a Servlet container, `Filter`s are invoked based upon the URL alone. However, `FilterChainProxy` can determine invocation based upon anything in the `HttpServletRequest` by leveraging the `RequestMatcher` interface.

In fact, `FilterChainProxy` can be used to determine which `SecurityFilterChain` should be used. This allows providing a totally separate configuration for different *slices* of your application.

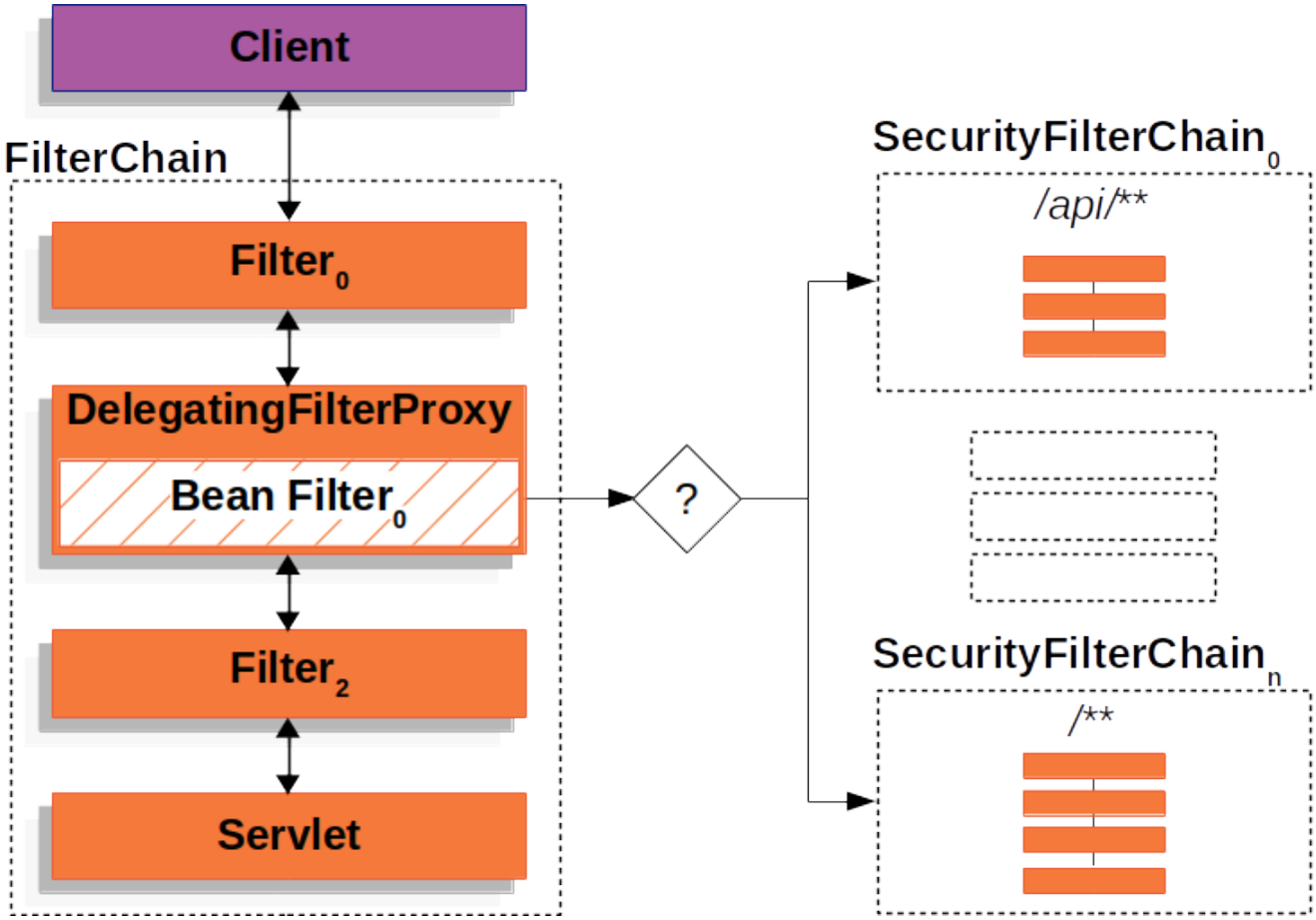


Figure 5. Multiple SecurityFilterChain

In the Multiple SecurityFilterChain Figure `FilterChainProxy` decides which `SecurityFilterChain` should be used. Only the first `SecurityFilterChain` that matches will be invoked. If a URL of `/api/messages/` is requested, it will first match on $SecurityFilterChain_0$'s pattern of `/api/**`, so only $SecurityFilterChain_0$ will be invoked even though it also matches on $SecurityFilterChain_n$. If a URL of `/messages/` is requested, it will not match on $SecurityFilterChain_0$'s pattern of `/api/**`, so `FilterChainProxy` will continue trying each `SecurityFilterChain`. Assuming that no other, `SecurityFilterChain` instances match $SecurityFilterChain_n$ will be invoked.

Notice that $SecurityFilterChain_0$ has only three security `Filter`s instances configured. However, $SecurityFilterChain_n$ has four security `Filter`s configured. It is important to note that each `SecurityFilterChain` can be unique and configured in isolation. In fact, a `SecurityFilterChain` might have zero security `Filter`s if the application wants Spring Security to ignore certain requests.

9.5. Security Filters

The Security Filters are inserted into the FilterChainProxy with the SecurityFilterChain API. The order of `Filter`s matters. It is typically not necessary to know the ordering of Spring Security's `Filter`s. However, there are times that it is beneficial to know the ordering

Below is a comprehensive list of Spring Security Filter ordering:

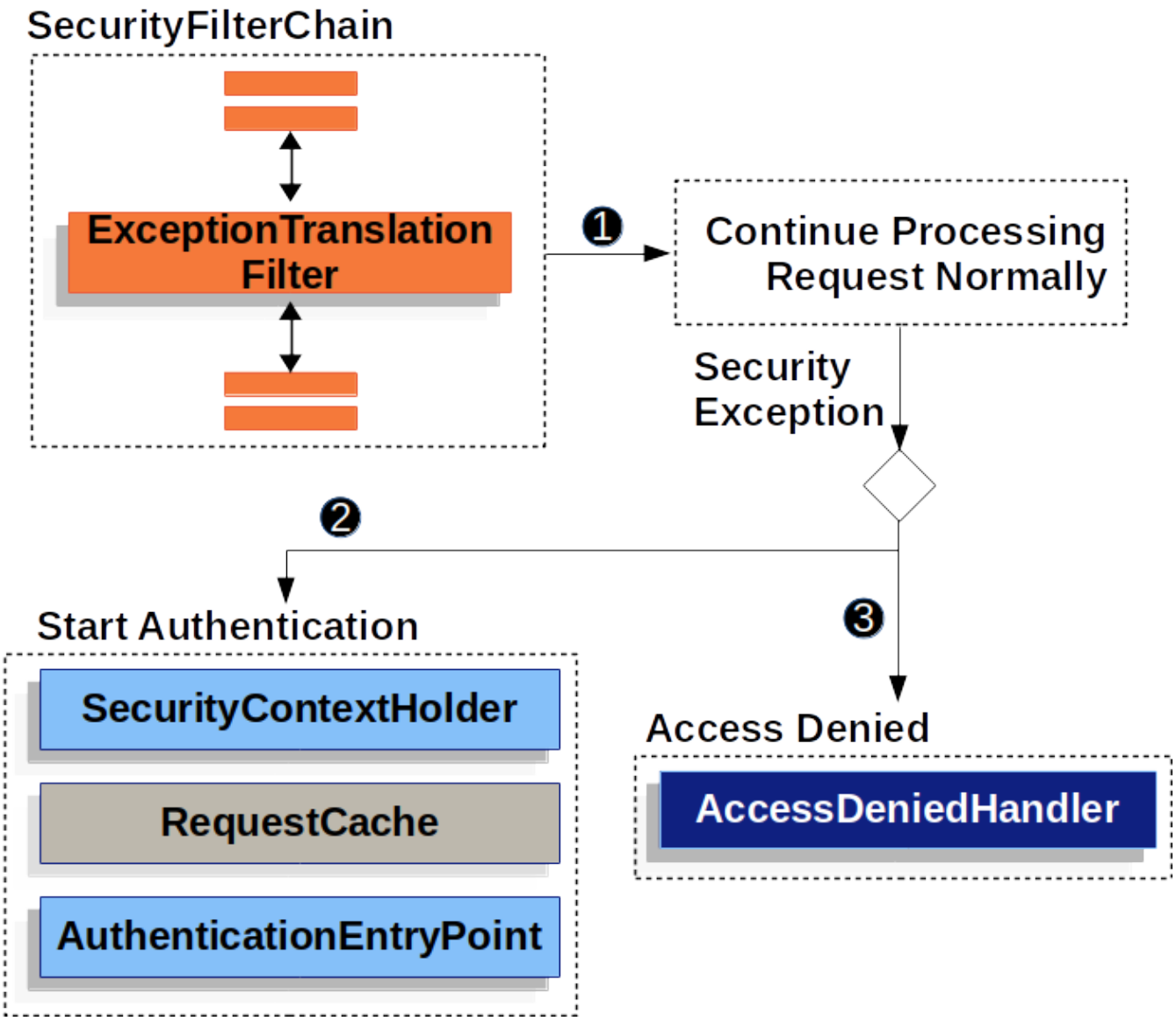
- ChannelProcessingFilter
- WebAsyncManagerIntegrationFilter
- SecurityContextPersistenceFilter

- HeaderWriterFilter
- CorsFilter
- CsrfFilter
- LogoutFilter
- OAuth2AuthorizationRequestRedirectFilter
- Saml2WebSsoAuthenticationRequestFilter
- X509AuthenticationFilter
- AbstractPreAuthenticatedProcessingFilter
- CasAuthenticationFilter
- OAuth2LoginAuthenticationFilter
- Saml2WebSsoAuthenticationFilter
- `UsernamePasswordAuthenticationFilter`
- OpenIDAuthenticationFilter
- DefaultLoginPageGeneratingFilter
- DefaultLogoutPageGeneratingFilter
- ConcurrentSessionFilter
- `DigestAuthenticationFilter`
- BearerTokenAuthenticationFilter
- `BasicAuthenticationFilter`
- RequestCacheAwareFilter
- SecurityContextHolderAwareRequestFilter
- JaasApiIntegrationFilter
- RememberMeAuthenticationFilter
- AnonymousAuthenticationFilter
- OAuth2AuthorizationCodeGrantFilter
- SessionManagementFilter
- `ExceptionTranslationFilter`
- `FilterSecurityInterceptor`
- SwitchUserFilter

9.6. Handling Security Exceptions

The `ExceptionTranslationFilter` allows translation of `AccessDeniedException` and `AuthenticationException` into HTTP responses.

`ExceptionTranslationFilter` is inserted into the FilterChainProxy as one of the Security Filters.



- 1 First, the `ExceptionTranslationFilter` invokes `FilterChain.doFilter(request, response)` to invoke the rest of the application.
- 2 If the user is not authenticated or it is an `AuthenticationException`, then *Start Authentication*.
  - The SecurityContextHolder is cleared out
  - The `HttpServletRequest` is saved in the `RequestCache`. When the user successfully authenticates, the `RequestCache` is used to replay the original request.
  - The `AuthenticationEntryPoint` is used to request credentials from the client. For example, it might redirect to a log in page or send a `WWW-Authenticate` header.
- 3 Otherwise if it is an `AccessDeniedException`, then *Access Denied*. The `AccessDeniedHandler` is invoked to handle access denied.

If the application does not throw an `AccessDeniedException` or an `AuthenticationException`, then `ExceptionTranslationFilter` does not do anything.

The pseudocode for `ExceptionTranslationFilter` looks something like this:

ExceptionTranslationFilter pseudocode

```
try {
    filterChain.doFilter(request, response); ①
} catch (AccessDeniedException | AuthenticationException ex) {
    if (!authenticated || ex instanceof AuthenticationException) {
        startAuthentication(); ②
    } else {
        accessDenied(); ③
    }
}
```

① You will recall from A Review of `Filters` that invoking `FilterChain.doFilter(request, response)` is the equivalent of invoking the rest of the application. This means that if another part of the application, (i.e. `FilterSecurityInterceptor` or method security) throws an `AuthenticationException` or `AccessDeniedException` it will be caught and handled here.

② If the user is not authenticated or it is an `AuthenticationException`, then *Start Authentication*.

③ Otherwise, *Access Denied*

Chapter 10. Authentication

Spring Security provides comprehensive support for Authentication. This section discusses:

Architecture Components

This section describes the main architectural components of Spring Security's used in Servlet authentication. If you need concrete flows that explain how these pieces fit together, look at the Authentication Mechanism specific sections.

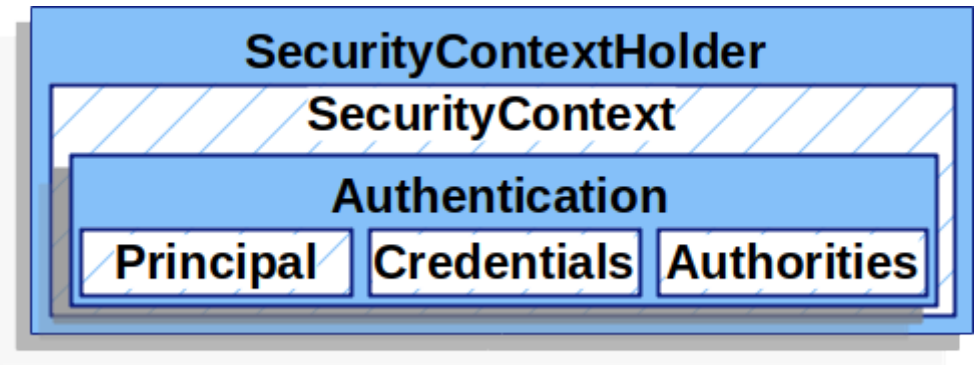
- SecurityContextHolder - The `SecurityContextHolder` is where Spring Security stores the details of who is authenticated.
- SecurityContext - is obtained from the `SecurityContextHolder` and contains the `Authentication` of the currently authenticated user.
- Authentication - Can be the input to `AuthenticationManager` to provide the credentials a user has provided to authenticate or the current user from the `SecurityContext`.
- GrantedAuthority - An authority that is granted to the principal on the `Authentication` (i.e. roles, scopes, etc.)
- AuthenticationManager - the API that defines how Spring Security's Filters perform authentication.
- ProviderManager - the most common implementation of `AuthenticationManager`.
- AuthenticationProvider - used by `ProviderManager` to perform a specific type of authentication.
- Request Credentials with `AuthenticationEntryPoint` - used for requesting credentials from a client (i.e. redirecting to a log in page, sending a `WWW-Authenticate` response, etc.)
- AbstractAuthenticationProcessingFilter - a base `Filter` used for authentication. This also gives a good idea of the high level flow of authentication and how pieces work together.

Authentication Mechanisms

- Username and Password - how to authenticate with a username/password
- OAuth 2.0 Login - OAuth 2.0 Log In with OpenID Connect and non-standard OAuth 2.0 Login (i.e. GitHub)
- SAML 2.0 Login - SAML 2.0 Log In
- Central Authentication Server (CAS) - Central Authentication Server (CAS) Support
- Remember Me - How to remember a user past session expiration
- JAAS Authentication - Authenticate with JAAS
- OpenID - OpenID Authentication (not to be confused with OpenID Connect)
- Pre-Authentication Scenarios - Authenticate with an external mechanism such as SiteMinder or Java EE security but still use Spring Security for authorization and protection against common exploits.
- X509 Authentication - X509 Authentication

10.1. SecurityContextHolder

At the heart of Spring Security's authentication model is the `SecurityContextHolder`. It contains the SecurityContext.



The `SecurityContextHolder` is where Spring Security stores the details of who is authenticated. Spring Security does not care how the `SecurityContextHolder` is populated. If it contains a value, then it is used as the currently authenticated user.

The simplest way to indicate a user is authenticated is to set the `SecurityContextHolder` directly.

Example 56. Setting `SecurityContextHolder`

Java

```
SecurityContext context = SecurityContextHolder.createEmptyContext(); ①
Authentication authentication =
    new TestingAuthenticationToken("username", "password", "ROLE_USER"); ②
context.setAuthentication(authentication);

SecurityContextHolder.setContext(context); ③
```

Kotlin

```
val context: SecurityContext = SecurityContextHolder.createEmptyContext() ①
val authentication: Authentication = TestingAuthenticationToken("username",
"password", "ROLE_USER") ②
context.authentication = authentication

SecurityContextHolder.setContext(context) ③
```

① We start by creating an empty `SecurityContext`. It is important to create a new `SecurityContext` instance instead of using `SecurityContextHolder.getContext().setAuthentication(authentication)` to avoid race conditions across multiple threads.

② Next we create a new `Authentication` object. Spring Security does not care what type of `Authentication` implementation is set on the `SecurityContext`. Here we use `TestingAuthenticationToken` because it is very simple. A more common production scenario is

`UsernamePasswordAuthenticationToken(userDetails, password, authorities)`.

③ Finally, we set the `SecurityContext` on the `SecurityContextHolder`. Spring Security will use this information for authorization.

If you wish to obtain information about the authenticated principal, you can do so by accessing the `SecurityContextHolder`.

Example 57. Access Currently Authenticated User

Java

```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
String username = authentication.getName();
Object principal = authentication.getPrincipal();
Collection<? extends GrantedAuthority> authorities =
authentication.getAuthorities();
```

Kotlin

```
val context = SecurityContextHolder.getContext()
val authentication = context.authentication
val username = authentication.name
val principal = authentication.principal
val authorities = authentication.authorities
```

By default the `SecurityContextHolder` uses a `ThreadLocal` to store these details, which means that the `SecurityContext` is always available to methods in the same thread, even if the `SecurityContext` is not explicitly passed around as an argument to those methods. Using a `ThreadLocal` in this way is quite safe if care is taken to clear the thread after the present principal's request is processed. Spring Security's FilterChainProxy ensures that the `SecurityContext` is always cleared.

Some applications aren't entirely suitable for using a `ThreadLocal`, because of the specific way they work with threads. For example, a Swing client might want all threads in a Java Virtual Machine to use the same security context. `SecurityContextHolder` can be configured with a strategy on startup to specify how you would like the context to be stored. For a standalone application you would use the `SecurityContextHolder.MODE_GLOBAL` strategy. Other applications might want to have threads spawned by the secure thread also assume the same security identity. This is achieved by using `SecurityContextHolder.MODE_INHERITABLETHREADLOCAL`. You can change the mode from the default `SecurityContextHolder.MODE_THREADLOCAL` in two ways. The first is to set a system property, the second is to call a static method on `SecurityContextHolder`. Most applications won't need to change from the default, but if you do, take a look at the JavaDoc for `SecurityContextHolder` to learn more.

10.2. SecurityContext

The `SecurityContext` is obtained from the SecurityContextHolder. The `SecurityContext` contains an Authentication object.

10.3. Authentication

The `Authentication` serves two main purposes within Spring Security:

- An input to `AuthenticationManager` to provide the credentials a user has provided to authenticate. When used in this scenario, `isAuthenticated()` returns `false`.
- Represents the currently authenticated user. The current `Authentication` can be obtained from the SecurityContext.

The `Authentication` contains:

- `principal` - identifies the user. When authenticating with a username/password this is often an instance of `UserDetails`.
- `credentials` - Often a password. In many cases this will be cleared after the user is authenticated to ensure it is not leaked.
- `authorities` - the `GrantedAuthority`s are high level permissions the user is granted. A few examples are roles or scopes.

10.4. GrantedAuthority

`GrantedAuthority`s are high level permissions the user is granted. A few examples are roles or scopes.

`GrantedAuthority`s can be obtained from the `Authentication.getAuthorities()` method. This method provides a `Collection` of `GrantedAuthority` objects. A `GrantedAuthority` is, not surprisingly, an authority that is granted to the principal. Such authorities are usually "roles", such as `ROLE_ADMINISTRATOR` or `ROLE_HR_SUPERVISOR`. These roles are later on configured for web authorization, method authorization and domain object authorization. Other parts of Spring Security are capable of interpreting these authorities, and expect them to be present. When using username/password based authentication `GrantedAuthority`s are usually loaded by the `UserDetailsService`.

Usually the `GrantedAuthority` objects are application-wide permissions. They are not specific to a given domain object. Thus, you wouldn't likely have a `GrantedAuthority` to represent a permission to `Employee` object number 54, because if there are thousands of such authorities you would quickly run out of memory (or, at the very least, cause the application to take a long time to authenticate a user). Of course, Spring Security is expressly designed to handle this common requirement, but you'd instead use the project's domain object security capabilities for this purpose.

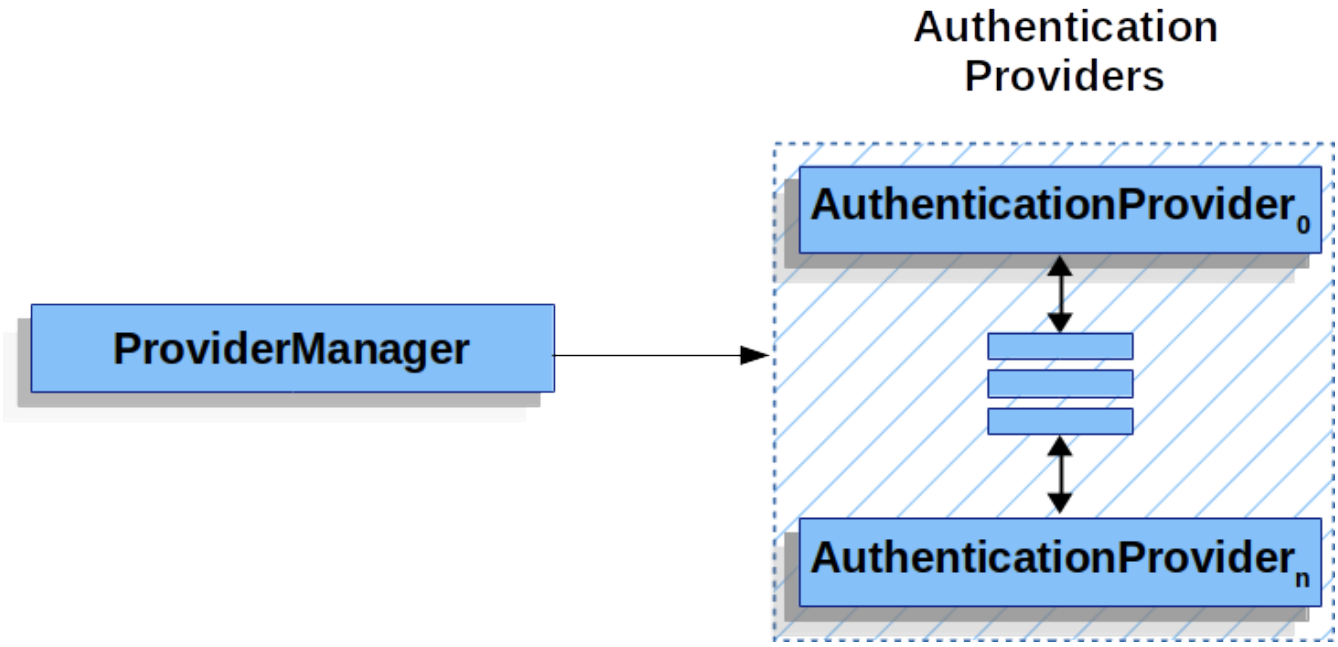
10.5. AuthenticationManager

`AuthenticationManager` is the API that defines how Spring Security's Filters perform authentication. The `Authentication` that is returned is then set on the SecurityContextHolder by the controller (i.e. Spring Security's `Filterss`) that invoked the `AuthenticationManager`. If you are not integrating with *Spring Security's* `Filters`s you can set the `SecurityContextHolder` directly and are not required to use an `AuthenticationManager`.

While the implementation of `AuthenticationManager` could be anything, the most common implementation is `ProviderManager`.

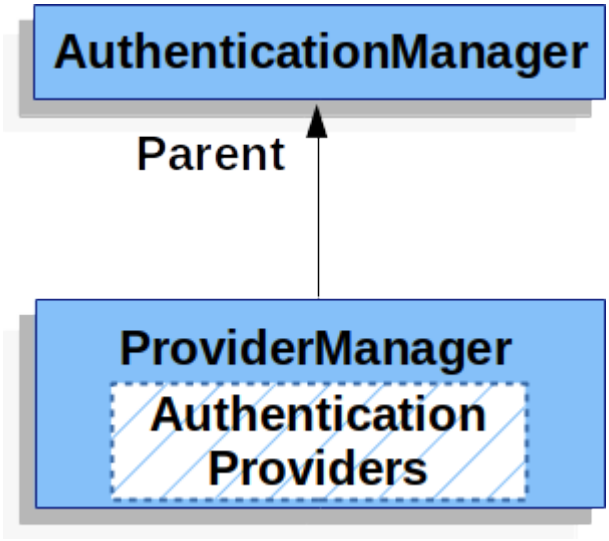
10.6. ProviderManager

`ProviderManager` is the most commonly used implementation of `AuthenticationManager`. `ProviderManager` delegates to a `List` of `AuthenticationProviders`. Each `AuthenticationProvider` has an opportunity to indicate that authentication should be successful, fail, or indicate it cannot make a decision and allow a downstream `AuthenticationProvider` to decide. If none of the configured `AuthenticationProviders` can authenticate, then authentication will fail with a `ProviderNotFoundException` which is a special `AuthenticationException` that indicates the `ProviderManager` was not configured to support the type of `Authentication` that was passed into it.

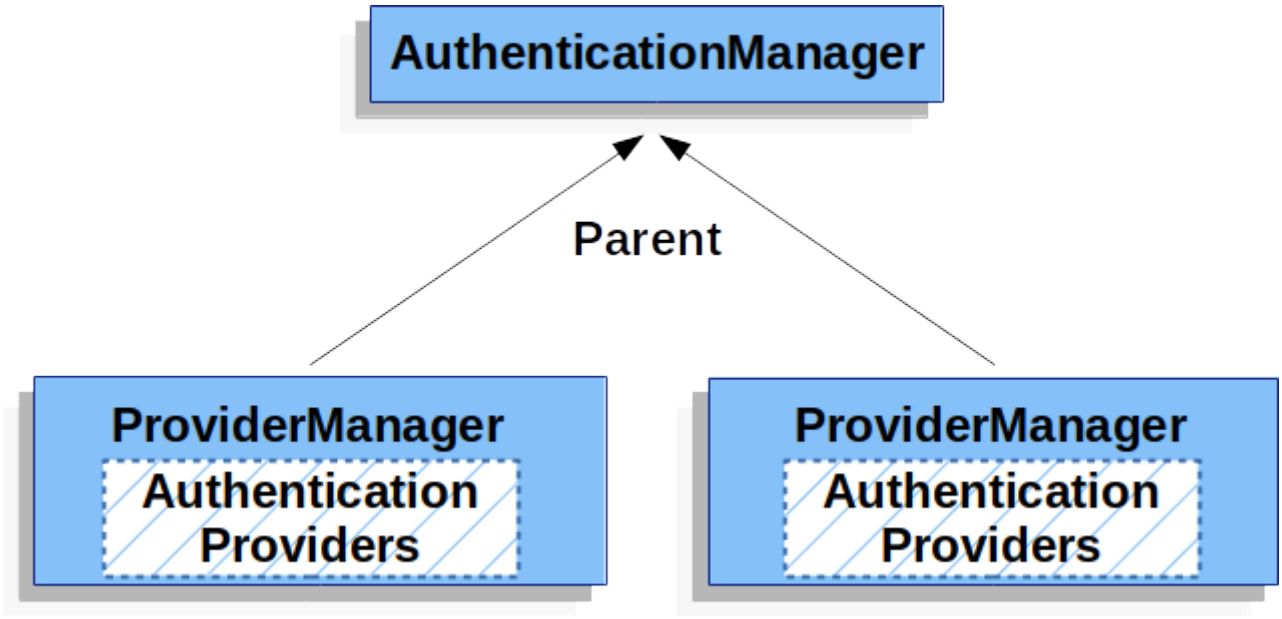


In practice each `AuthenticationProvider` knows how to perform a specific type of authentication. For example, one `AuthenticationProvider` might be able to validate a username/password, while another might be able to authenticate a SAML assertion. This allows each `AuthenticationProvider` to do a very specific type of authentication, while supporting multiple types of authentication and only exposing a single `AuthenticationManager` bean.

`ProviderManager` also allows configuring an optional parent `AuthenticationManager` which is consulted in the event that no `AuthenticationProvider` can perform authentication. The parent can be any type of `AuthenticationManager`, but it is often an instance of `ProviderManager`.



In fact, multiple `ProviderManager` instances might share the same parent `AuthenticationManager`. This is somewhat common in scenarios where there are multiple `SecurityFilterChain` instances that have some authentication in common (the shared parent `AuthenticationManager`), but also different authentication mechanisms (the different `ProviderManager` instances).



By default `ProviderManager` will attempt to clear any sensitive credentials information from the `Authentication` object which is returned by a successful authentication request. This prevents information like passwords being retained longer than necessary in the `HttpSession`.

This may cause issues when you are using a cache of user objects, for example, to improve performance in a stateless application. If the `Authentication` contains a reference to an object in the cache (such as a `UserDetails` instance) and this has its credentials removed, then it will no longer be possible to authenticate against the cached value. You need to take this into account if you are using a cache. An obvious solution is to make a copy of the object first, either in the cache implementation or in the `AuthenticationProvider` which creates the returned `Authentication` object. Alternatively, you can disable the `eraseCredentialsAfterAuthentication` property on `ProviderManager`. See the Javadoc for more information.

10.7. AuthenticationProvider

Multiple `AuthenticationProviders` can be injected into `ProviderManager`. Each `AuthenticationProvider` performs a specific type of authentication. For example, `DaoAuthenticationProvider` supports username/password based authentication while `JwtAuthenticationProvider` supports authenticating a JWT token.

10.8. Request Credentials with `AuthenticationEntryPoint`

`AuthenticationEntryPoint` is used to send an HTTP response that requests credentials from a client.

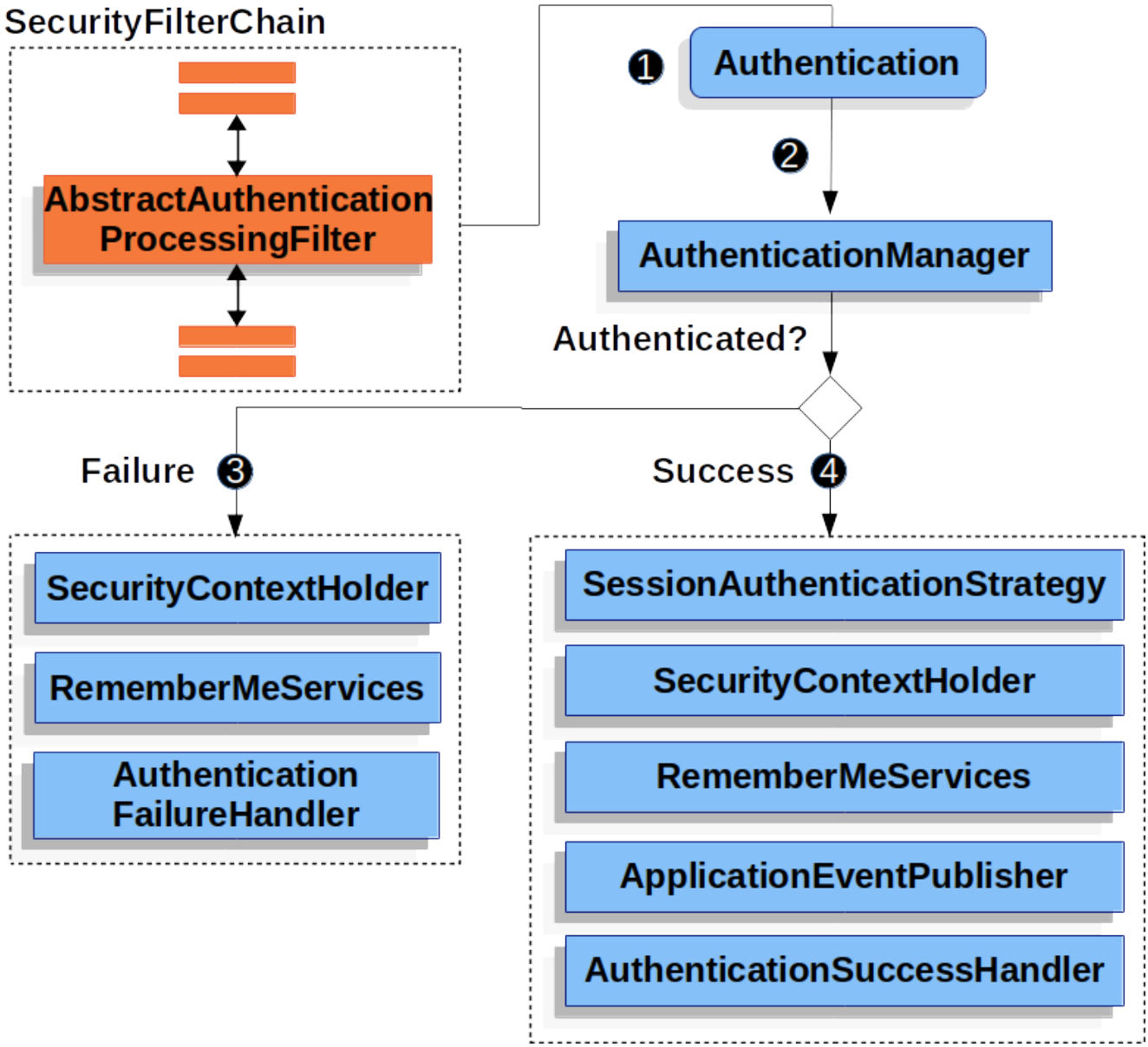
Sometimes a client will proactively include credentials such as a username/password to request a resource. In these cases, Spring Security does not need to provide an HTTP response that requests credentials from the client since they are already included.

In other cases, a client will make an unauthenticated request to a resource that they are not authorized to access. In this case, an implementation of `AuthenticationEntryPoint` is used to request credentials from the client. The `AuthenticationEntryPoint` implementation might perform a redirect to a log in page, respond with an WWW-Authenticate header, etc.

10.9. AbstractAuthenticationProcessingFilter

`AbstractAuthenticationProcessingFilter` is used as a base `Filter` for authenticating a user's credentials. Before the credentials can be authenticated, Spring Security typically requests the credentials using `AuthenticationEntryPoint`.

Next, the `AbstractAuthenticationProcessingFilter` can authenticate any authentication requests that are submitted to it.



1 When the user submits their credentials, the `AbstractAuthenticationProcessingFilter` creates an `Authentication` from the `HttpServletRequest` to be authenticated. The type of `Authentication` created depends on the subclass of `AbstractAuthenticationProcessingFilter`. For example, `UsernamePasswordAuthenticationFilter` creates a `UsernamePasswordAuthenticationToken` from a *username* and *password* that are submitted in the `HttpServletRequest`.

2 Next, the `Authentication` is passed into the `AuthenticationManager` to be authenticated.

3 If authentication fails, then *Failure*

- The SecurityContextHolder is cleared out.
- `RememberMeServices.loginFail` is invoked. If remember me is not configured, this is a no-op.
- `AuthenticationFailureHandler` is invoked.

4 If authentication is successful, then *Success*.

- `SessionAuthenticationStrategy` is notified of a new log in.
- The Authentication is set on the SecurityContextHolder. Later the

`SecurityContextPersistenceFilter` saves the `SecurityContext` to the `HttpSession`.

- `RememberMeServices.loginSuccess` is invoked. If remember me is not configured, this is a no-op.
- `ApplicationEventPublisher` publishes an `InteractiveAuthenticationSuccessEvent`.
- `AuthenticationSuccessHandler` is invoked.

10.10. Username/Password Authentication

One of the most common ways to authenticate a user is by validating a username and password. As such, Spring Security provides comprehensive support for authenticating with a username and password.

Reading the Username & Password

Spring Security provides the following built in mechanisms for reading a username and password from the `HttpServletRequest`:

- Form Login
- Basic Authentication
- Digest Authentication

Storage Mechanisms

Each of the supported mechanisms for reading a username and password can leverage any of the supported storage mechanisms:

- Simple Storage with In-Memory Authentication
- Relational Databases with JDBC Authentication
- Custom data stores with UserDetailsService
- LDAP storage with LDAP Authentication

10.10.1. Form Login

Spring Security provides support for username and password being provided through an html form. This section provides details on how form based authentication works within Spring Security.

Let's take a look at how form based log in works within Spring Security. First, we see how the user is redirected to the log in form.

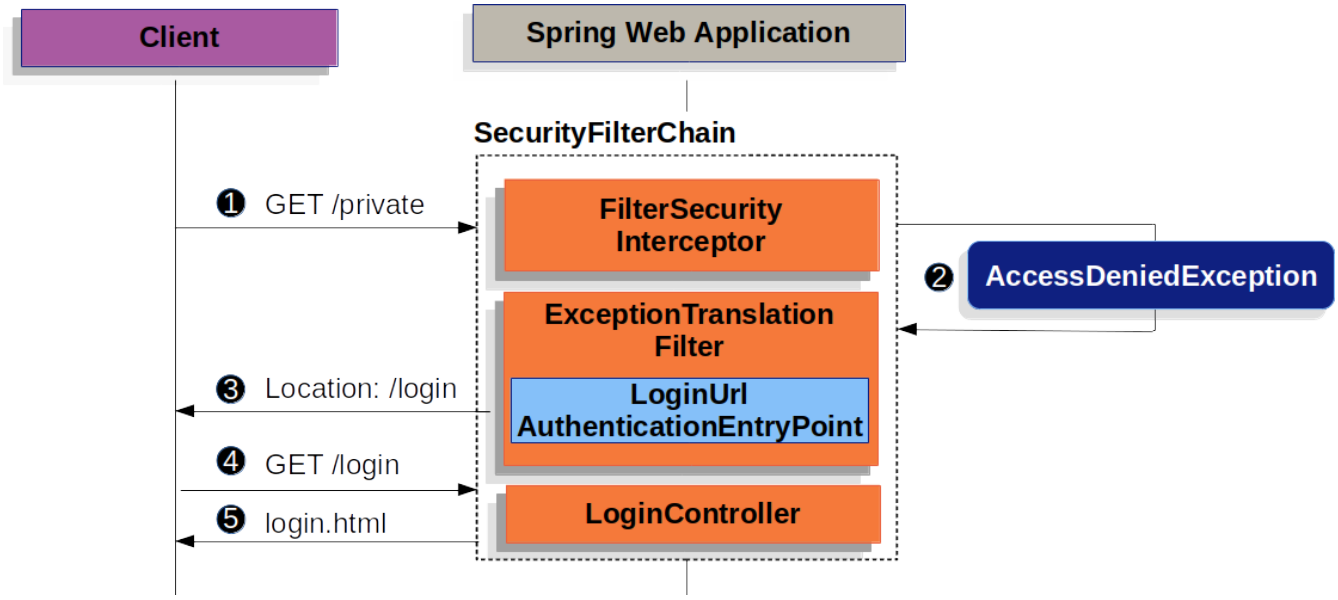


Figure 6. Redirecting to the Log In Page

The figure builds off our `SecurityFilterChain` diagram.

1 First, a user makes an unauthenticated request to the resource `/private` for which it is not authorized.

2 Spring Security's `FilterSecurityInterceptor` indicates that the unauthenticated request is *Denied* by throwing an `AccessDeniedException`.

3 Since the user is not authenticated, `ExceptionTranslationFilter` initiates *Start Authentication* and sends a redirect to the log in page with the configured `AuthenticationEntryPoint`. In most cases the `AuthenticationEntryPoint` is an instance of `LoginUrlAuthenticationEntryPoint`.

4 The browser will then request the log in page that it was redirected to.

5 Something within the application, must render the log in page.

When the username and password are submitted, the `UsernamePasswordAuthenticationFilter` authenticates the username and password. The `UsernamePasswordAuthenticationFilter` extends AbstractAuthenticationProcessingFilter, so this diagram should look pretty similar.

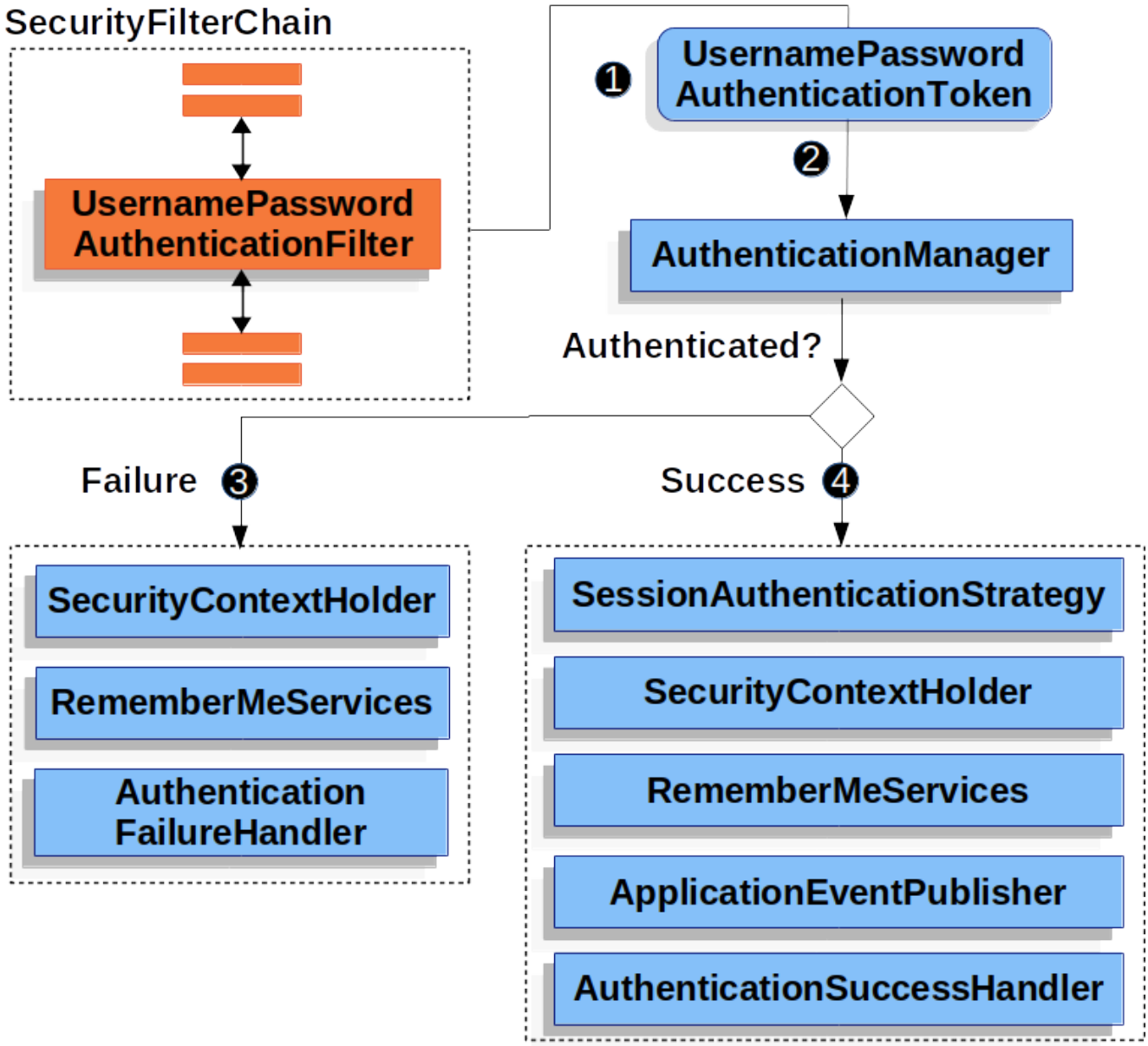


Figure 7. Authenticating Username and Password

The figure builds off our `SecurityFilterChain` diagram.

1 When the user submits their username and password, the `UsernamePasswordAuthenticationFilter` creates a `UsernamePasswordAuthenticationToken` which is a type of `Authentication` by extracting the username and password from the `HttpServletRequest`.

2 Next, the `UsernamePasswordAuthenticationToken` is passed into the `AuthenticationManager` to be authenticated. The details of what `AuthenticationManager` look like depend on how the user information is stored.

3 If authentication fails, then *Failure*

- The SecurityContextHolder is cleared out.
- `RememberMeServices.loginFail` is invoked. If remember me is not configured, this is a no-op.
- `AuthenticationFailureHandler` is invoked.

4 If authentication is successful, then *Success*.

- `SessionAuthenticationStrategy` is notified of a new log in.
- The Authentication is set on the SecurityContextHolder.
- `RememberMeServices.loginSuccess` is invoked. If remember me is not configured, this is a no-op.
- `ApplicationEventPublisher` publishes an `InteractiveAuthenticationSuccessEvent`.
- The `AuthenticationSuccessHandler` is invoked. Typically this is a `SimpleUrlAuthenticationSuccessHandler` which will redirect to a request saved by `ExceptionTranslationFilter` when we redirect to the log in page.

Spring Security form log in is enabled by default. However, as soon as any servlet based configuration is provided, form based log in must be explicitly provided. A minimal, explicit Java configuration can be found below:

Example 58. Form Log In

Java

```
protected void configure(HttpSecurity http) {
    http
        // ...
        .formLogin(withDefaults());
}
```

XML

```
<http>
    <!-- ... -->
    <form-login />
</http>
```

Kotlin

```
fun configure(http: HttpSecurity) {
    http {
        // ...
        formLogin { }
    }
}
```

In this configuration Spring Security will render a default log in page. Most production applications will require a custom log in form.

The configuration below demonstrates how to provide a custom log in form.

Example 59. Custom Log In Form Configuration

Java

```
protected void configure(HttpSecurity http) throws Exception {
    http
        // ...
        .formLogin(form -> form
            .loginPage("/login")
            .permitAll()
        );
}
```

XML

```
<http>
    <!-- ... -->
    <intercept-url pattern="/login" access="permitAll" />
    <form-login login-page="/login" />
</http>
```

Kotlin

```
fun configure(http: HttpSecurity) {
    http {
        // ...
        formLogin {
            loginPage = "/login"
            permitAll()
        }
    }
}
```

When the login page is specified in the Spring Security configuration, you are responsible for rendering the page. Below is a Thymeleaf template that produces an HTML login form that complies with a login page of `/login`.:

Example 60. Log In Form

src/main/resources/templates/login.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org">
    <head>
        <title>Please Log In</title>
    </head>
    <body>
        <h1>Please Log In</h1>
        <div th:if="${param.error}">
            Invalid username and password.</div>
        <div th:if="${param.logout}">
            You have been logged out.</div>
        <form th:action="@{/login}" method="post">
            <div>
            <input type="text" name="username" placeholder="Username"/>
            </div>
            <div>
            <input type="password" name="password" placeholder="Password"/>
            </div>
            <input type="submit" value="Log in" />
        </form>
    </body>
</html>
```

There are a few key points about the default HTML form:

- The form should perform a `post` to `/login`
- The form will need to include a CSRF Token which is automatically included by Thymeleaf.
- The form should specify the username in a parameter named `username`
- The form should specify the password in a parameter named `password`
- If the HTTP parameter error is found, it indicates the user failed to provide a valid username / password
- If the HTTP parameter logout is found, it indicates the user has logged out successfully

Many users will not need much more than to customize the log in page. However, if needed everything above can be customized with additional configuration.

If you are using Spring MVC, you will need a controller that maps `GET /login` to the login template we created. A minimal sample `LoginController` can be see below:

Example 61. LoginController

Java

```
@Controller
class LoginController {
    @GetMapping("/login")
    String login() {
        return "login";
    }
}
```

Kotlin

```
@Controller
class LoginController {
    @GetMapping("/login")
    fun login(): String {
        return "login"
    }
}
```

10.10.2. Basic Authentication

This section provides details on how Spring Security provides support for Basic HTTP Authentication for servlet based applications.

Let's take a look at how HTTP Basic Authentication works within Spring Security. First, we see the WWW-Authenticate header is sent back to an unauthenticated client.

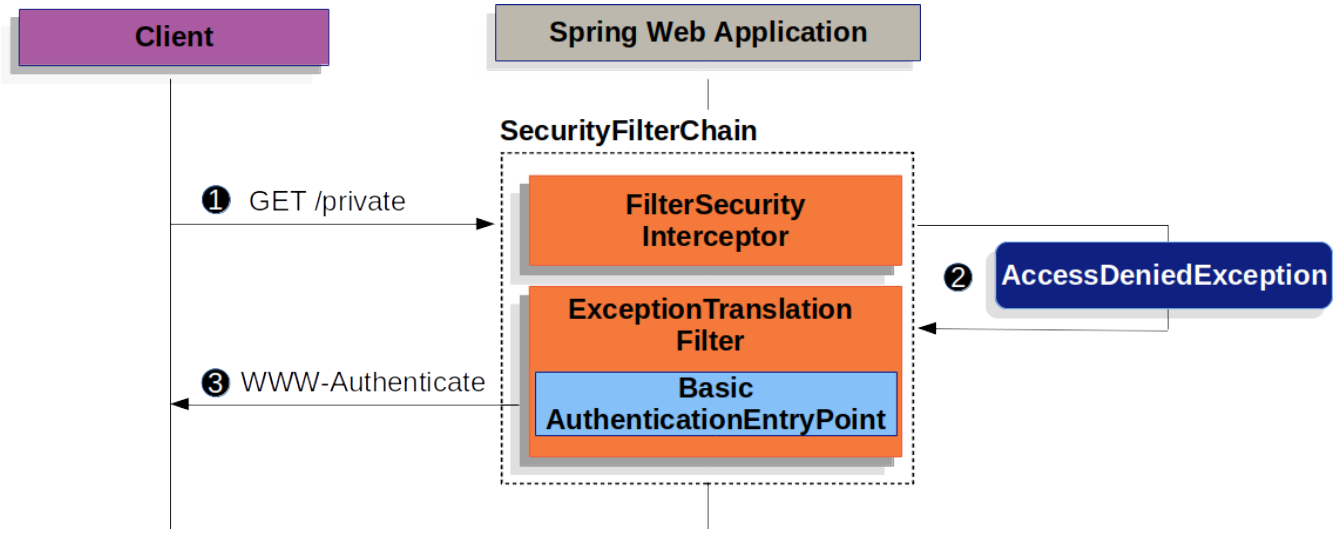


Figure 8. Sending WWW-Authenticate Header

The figure builds off our `SecurityFilterChain` diagram.

1. First, a user makes an unauthenticated request to the resource `/private` for which it is not

authorized.

❷ Spring Security's `FilterSecurityInterceptor` indicates that the unauthenticated request is *Denied* by throwing an `AccessDeniedException`.

❸ Since the user is not authenticated, `ExceptionTranslationFilter` initiates *Start Authentication*. The configured `AuthenticationEntryPoint` is an instance of `BasicAuthenticationEntryPoint` which sends a WWW-Authenticate header. The `RequestCache` is typically a `NullRequestCache` that does not save the request since the client is capable of replaying the requests it originally requested.

When a client receives the WWW-Authenticate header it knows it should retry with a username and password. Below is the flow for the username and password being processed.

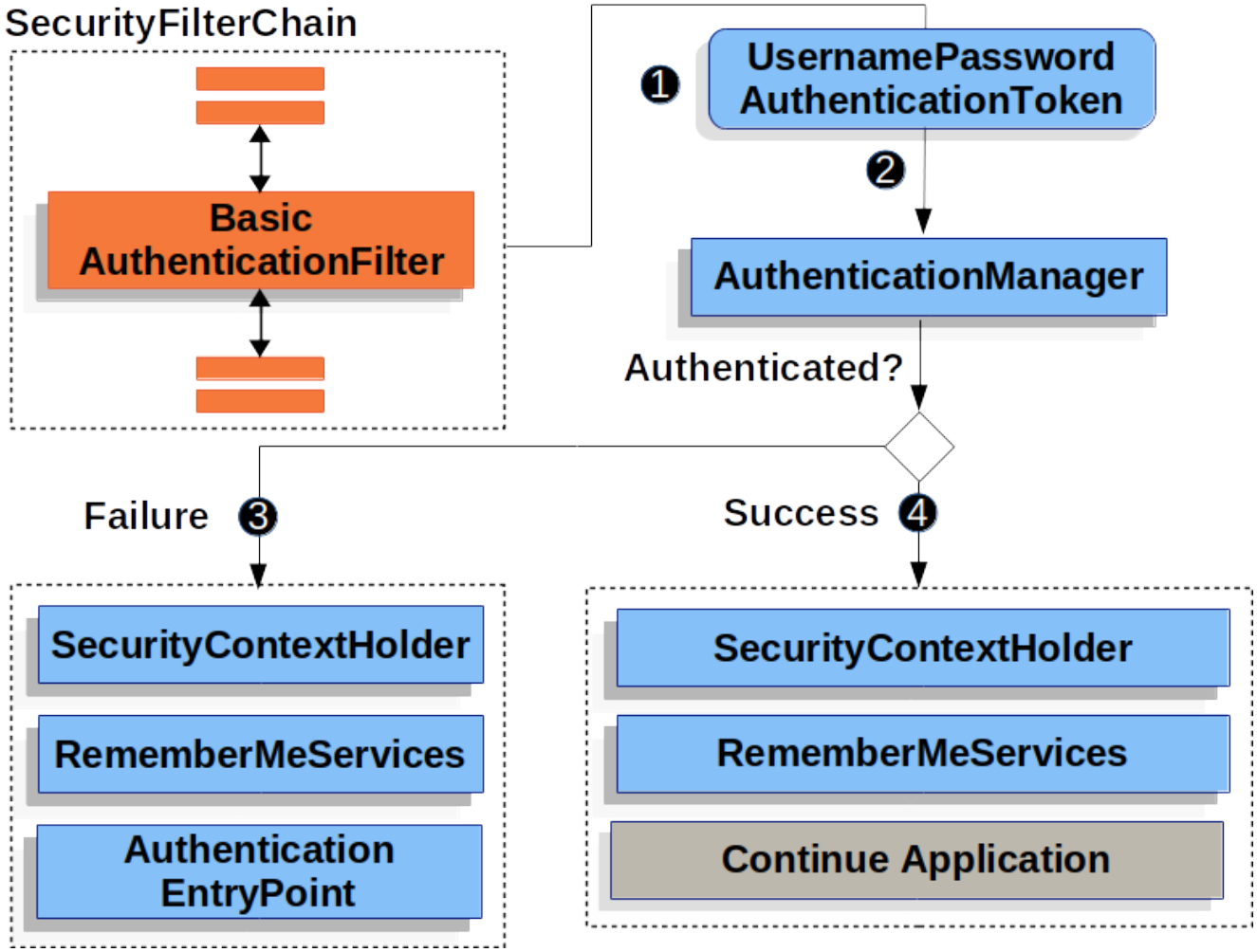


Figure 9. Authenticating Username and Password

The figure builds off our `SecurityFilterChain` diagram.

❶ When the user submits their username and password, the `BasicAuthenticationFilter` creates a `UsernamePasswordAuthenticationToken` which is a type of `Authentication` by extracting the username and password from the `HttpServletRequest`.

❷ Next, the `UsernamePasswordAuthenticationToken` is passed into the `AuthenticationManager` to be authenticated. The details of what `AuthenticationManager` look like depend on how the user information is stored.

3 If authentication fails, then *Failure*

- The SecurityContextHolder is cleared out.
- `RememberMeServices.loginFail` is invoked. If remember me is not configured, this is a no-op.
- `AuthenticationEntryPoint` is invoked to trigger the WWW-Authenticate to be sent again.

4 If authentication is successful, then *Success*.

- The Authentication is set on the SecurityContextHolder.
- `RememberMeServices.loginSuccess` is invoked. If remember me is not configured, this is a no-op.
- The `BasicAuthenticationFilter` invokes `FilterChain.doFilter(request,response)` to continue with the rest of the application logic.

Spring Security's HTTP Basic Authentication support in is enabled by default. However, as soon as any servlet based configuration is provided, HTTP Basic must be explicitly provided.

A minimal, explicit configuration can be found below:

Example 62. Explicit HTTP Basic Configuration

Java

```
protected void configure(HttpSecurity http) {
    http
        // ...
        .httpBasic(withDefaults());
}
```

XML

```
<http>
    <!-- ... -->
    <http-basic />
</http>
```

Kotlin

```
fun configure(http: HttpSecurity) {
    http {
        // ...
        httpBasic { }
    }
}
```

10.10.3. Digest Authentication

This section provides details on how Spring Security provides support for Digest Authentication which is provided `DigestAuthenticationFilter`.

You should not use Digest Authentication in modern applications because it is not considered secure. The most obvious problem is that you must store your passwords in plaintext, encrypted, or an MD5 format. All of these storage formats are considered insecure. Instead, you should store credentials using a one way adaptive password hash (i.e. bCrypt, PBKDF2, SCrypt, etc) which is not supported by Digest Authentication.

Digest Authentication attempts to solve many of the weaknesses of Basic authentication, specifically by ensuring credentials are never sent in clear text across the wire. Many browsers support Digest Authentication.

The standard governing HTTP Digest Authentication is defined by RFC 2617, which updates an earlier version of the Digest Authentication standard prescribed by RFC 2069. Most user agents implement RFC 2617. Spring Security's Digest Authentication support is compatible with the "auth" quality of protection (`qop`) prescribed by RFC 2617, which also provides backward compatibility with RFC 2069. Digest Authentication was seen as a more attractive option if you need to use unencrypted HTTP (i.e. no TLS/HTTPS) and wish to maximise security of the authentication process. However, everyone should use HTTPS.

Central to Digest Authentication is a "nonce". This is a value the server generates. Spring Security's nonce adopts the following format:

Example 63. Digest Syntax

```
base64(expirationTime + ":" + md5Hex(expirationTime + ":" + key))
expirationTime:   The date and time when the nonce expires, expressed in
milliseconds
key:              A private key to prevent modification of the nonce token
```

You will need to ensure you configure insecure plain text Password Storage using NoOpPasswordEncoder`. The following provides an example of configuring Digest Authentication with Java Configuration:

Example 64. Digest Authentication

Java

```
@Autowired
UserDetailsService userDetailsService;

DigestAuthenticationEntryPoint entryPoint() {
    DigestAuthenticationEntryPoint result = new DigestAuthenticationEntryPoint();
    result.setRealmName("My App Relam");
    result.setKey("3028472b-da34-4501-bfd8-a355c42bdf92");
}

DigestAuthenticationFilter digestAuthenticationFilter() {
    DigestAuthenticationFilter result = new DigestAuthenticationFilter();
    result.setUserDetailsService(userDetailsService);
    result.setAuthenticationEntryPoint(entryPoint());
}

protected void configure(HttpSecurity http) throws Exception {
    http
        // ...
        .exceptionHandling(e ->
e.authenticationEntryPoint(authenticationEntryPoint()))
        .addFilterBefore(digestFilter());
}
```

XML

```
<b:bean id="digestFilter"

class="org.springframework.security.web.authentication.www.DigestAuthenticationFil
ter"
    p:userDetailsService-ref="jdbcDaoImpl"
    p:authenticationEntryPoint-ref="digestEntryPoint"
/>

<b:bean id="digestEntryPoint"

class="org.springframework.security.web.authentication.www.DigestAuthenticationEnt
ryPoint"
    p:realmName="My App Realm"
    p:key="3028472b-da34-4501-bfd8-a355c42bdf92"
/>

<http>
    <!-- ... -->
    <custom-filter ref="userFilter" position="DIGEST_AUTH_FILTER"/>
</http>
```

10.10.4. In-Memory Authentication

Spring Security's `InMemoryUserDetailsManager` implements UserDetailsService to provide support for username/password based authentication that is retrieved in memory. `InMemoryUserDetailsManager` provides management of `UserDetails` by implementing the `UserDetailsManager` interface. `UserDetails` based authentication is used by Spring Security when it is configured to accept a username/password for authentication.

In this sample we use Spring Boot CLI to encode the password of `password` and get the encoded password of `{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW`.

Example 65. InMemoryUserDetailsManager Java Configuration

Java

```
@Bean
public UserDetailsService users() {
    UserDetails user = User.builder()
        .username("user")

.password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER")
        .build();
    UserDetails admin = User.builder()
        .username("admin")

.password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER", "ADMIN")
        .build();
    return new InMemoryUserDetailsManager(user, admin);
}
```

XML

```
<user-service>
    <user name="user"

password="{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW"
        authorities="ROLE_USER" />
    <user name="admin"

password="{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW"
        authorities="ROLE_USER,ROLE_ADMIN" />
</user-service>
```

Kotlin

```
@Bean
fun users(): UserDetailsService {
    val user = User.builder()
        .username("user")

.password("{bcrypt}$2a$10\$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER")
        .build()
    val admin = User.builder()
        .username("admin")

.password("{bcrypt}$2a$10\$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER", "ADMIN")
        .build()
    return InMemoryUserDetailsManager(user, admin)
}
```

The samples above store the passwords in a secure format, but leave a lot to be desired in terms of getting started experience.

In the sample below we leverage User.withDefaultPasswordEncoder to ensure that the password stored in memory is protected. However, it does not protect against obtaining the password by decompiling the source code. For this reason, `User.withDefaultPasswordEncoder` should only be used for "getting started" and is not intended for production.

Example 66. InMemoryUserDetailsManager with User.withDefaultPasswordEncoder

Java

```
@Bean
public UserDetailsService users() {
    // The builder will ensure the passwords are encoded before saving in memory
    UserBuilder users = User.withDefaultPasswordEncoder();
    UserDetails user = users
        .username("user")
        .password("password")
        .roles("USER")
        .build();
    UserDetails admin = users
        .username("admin")
        .password("password")
        .roles("USER", "ADMIN")
        .build();
    return new InMemoryUserDetailsManager(user, admin);
}
```

Kotlin

```
@Bean
fun users(): UserDetailsService {
    // The builder will ensure the passwords are encoded before saving in memory
    val users = User.withDefaultPasswordEncoder()
    val user = users
        .username("user")
        .password("password")
        .roles("USER")
        .build()
    val admin = users
        .username("admin")
        .password("password")
        .roles("USER", "ADMIN")
        .build()
    return InMemoryUserDetailsManager(user, admin)
}
```

There is no simple way to use `User.withDefaultPasswordEncoder` with XML based configuration. For demos or just getting started, you can choose to prefix the password with `{noop}` to indicate no encoding should be used.

Example 67. <user-service> {noop} XML Configuration

```
<user-service>
    <user name="user"
        password="{noop}password"
        authorities="ROLE_USER" />
    <user name="admin"
        password="{noop}password"
        authorities="ROLE_USER,ROLE_ADMIN" />
</user-service>
```

10.10.5. JDBC Authentication

Spring Security's `JdbcDaoImpl` implements `UserDetailsService` to provide support for username/password based authentication that is retrieved using JDBC. `JdbcUserDetailsManager` extends `JdbcDaoImpl` to provide management of `UserDetails` through the `UserDetailsManager` interface. `UserDetails` based authentication is used by Spring Security when it is configured to accept a username/password for authentication.

In the following sections we will discuss:

- The Default Schema used by Spring Security JDBC Authentication
- Setting up a DataSource
- JdbcUserDetailsManager Bean

Default Schema

Spring Security provides default queries for JDBC based authentication. This section provides the corresponding default schemas used with the default queries. You will need to adjust the schema to match any customizations to the queries and the database dialect you are using.

User Schema

`JdbcDaoImpl` requires tables to load the password, account status (enabled or disabled) and a list of authorities (roles) for the user. The default schema required can be found below.

The default schema is also exposed as a classpath resource named `org/springframework/security/core/userdetails/jdbc/users.ddl`.

Example 68. Default User Schema

```
create table users(
    username varchar_ignorecase(50) not null primary key,
    password varchar_ignorecase(500) not null,
    enabled boolean not null
);

create table authorities (
    username varchar_ignorecase(50) not null,
    authority varchar_ignorecase(50) not null,
    constraint fk_authorities_users foreign key(username) references
users(username)
);
create unique index ix_auth_username on authorities (username,authority);
```

Oracle is a popular database choice, but requires a slightly different schema. You can find the default Oracle Schema for users below.

Example 69. Default User Schema for Oracle Databases

```
CREATE TABLE USERS (
    USERNAME NVARCHAR2(128) PRIMARY KEY,
    PASSWORD NVARCHAR2(128) NOT NULL,
    ENABLED CHAR(1) CHECK (ENABLED IN ('Y','N') ) NOT NULL
);


CREATE TABLE AUTHORITIES (
    USERNAME NVARCHAR2(128) NOT NULL,
    AUTHORITY NVARCHAR2(128) NOT NULL
);
ALTER TABLE AUTHORITIES ADD CONSTRAINT AUTHORITIES_UNIQUE UNIQUE (USERNAME,
AUTHORITY);
ALTER TABLE AUTHORITIES ADD CONSTRAINT AUTHORITIES_FK1 FOREIGN KEY (USERNAME)
REFERENCES USERS (USERNAME) ENABLE;
```

Group Schema

If your application is leveraging groups, you will need to provide the groups schema. The default schema for groups can be found below.

Example 70. Default Group Schema

```
create table groups (
    id bigint generated by default as identity(start with 0) primary key,
    group_name varchar_ignorecase(50) not null
);

create table group_authorities (
    group_id bigint not null,
    authority varchar(50) not null,
    constraint fk_group_authorities_group foreign key(group_id) references
groups(id)
);

create table group_members (
    id bigint generated by default as identity(start with 0) primary key,
    username varchar(50) not null,
    group_id bigint not null,
    constraint fk_group_members_group foreign key(group_id) references groups(id)
);
```

Setting up a DataSource

Before we configure `JdbcUserDetailsManager`, we must create a `DataSource`. In our example, we will setup an embedded DataSource that is initialized with the default user schema.

Example 71. Embedded Data Source

Java

```
@Bean
DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(H2)

.addScript("classpath:org/springframework/security/core/userdetails/jdbc/users.ddl
")
        .build();
}
```

XML

```
<jdbc:embedded-database>
    <jdbc:script
location="classpath:org/springframework/security/core/userdetails/jdbc/users.ddl"/
>
</jdbc:embedded-database>
```

Kotlin

```
@Bean
fun dataSource(): DataSource {
    return EmbeddedDatabaseBuilder()
        .setType(H2)

.addScript("classpath:org/springframework/security/core/userdetails/jdbc/users.ddl
")
        .build()
}
```

In a production environment, you will want to ensure you setup a connection to an external database.

JdbcUserDetailsManager Bean

In this sample we use Spring Boot CLI to encode the password of `password` and get the encoded password of `{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW`. See the PasswordEncoder section for more details about how to store passwords.

Example 72. JdbcUserDetailsManager

Java

```
@Bean
UserDetailsManager users(DataSource dataSource) {
    UserDetails user = User.builder()
        .username("user")

.password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER")
        .build();
    UserDetails admin = User.builder()
        .username("admin")

.password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER", "ADMIN")
        .build();
    JdbcUserDetailsManager users = new JdbcUserDetailsManager(dataSource);
    users.createUser(user);
    users.createUser(admin);
    return users;
}
```

XML

```
<jdbc-user-service>
    <user name="user"

password="{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW"
        authorities="ROLE_USER" />
    <user name="admin"

password="{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW"
        authorities="ROLE_USER,ROLE_ADMIN" />
</jdbc-user-service>
```

Kotlin

```
@Bean
fun users(dataSource: DataSource): UserDetailsManager {
    val user = User.builder()
            .username("user")

.password("{bcrypt}$2a$10\$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
            .roles("USER")
            .build();
    val admin = User.builder()
            .username("admin")

.password("{bcrypt}$2a$10\$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
            .roles("USER", "ADMIN")
            .build();
    val users = JdbcUserDetailsManager(dataSource)
    users.createUser(user)
    users.createUser(admin)
    return users
}
```

10.10.6. UserDetails

`UserDetails` is returned by the `UserDetailsService`. The `DaoAuthenticationProvider` validates the `UserDetails` and then returns an `Authentication` that has a principal that is the `UserDetails` returned by the configured `UserDetailsService`.

10.10.7. UserDetailsService

`UserDetailsService` is used by `DaoAuthenticationProvider` for retrieving a username, password, and other attributes for authenticating with a username and password. Spring Security provides in-memory and JDBC implementations of `UserDetailsService`.

You can define custom authentication by exposing a custom `UserDetailsService` as a bean. For example, the following will customize authentication assuming that `CustomUserDetailsService` implements `UserDetailsService`:

This is only used if the `AuthenticationManagerBuilder` has not been populated and no `AuthenticationProviderBean` is defined.

Example 73. Custom UserDetailsService Bean

Java

```
@Bean
CustomUserDetailsService customUserDetailsService() {
    return new CustomUserDetailsService();
}
```

XML

```
<b:bean class="example.CustomUserDetailsService"/>
```

Kotlin

```
@Bean
fun customUserDetailsService() = CustomUserDetailsService()
```

10.10.8. PasswordEncoder

Spring Security's servlet support storing passwords securely by integrating with `PasswordEncoder`. Customizing the `PasswordEncoder` implementation used by Spring Security can be done by exposing a `PasswordEncoder` Bean.

10.10.9. DaoAuthenticationProvider

`DaoAuthenticationProvider` is an `AuthenticationProvider` implementation that leverages a `UserDetailsService` and `PasswordEncoder` to authenticate a username and password.

Let's take a look at how `DaoAuthenticationProvider` works within Spring Security. The figure explains details of how the `AuthenticationManager` in figures from Reading the Username & Password works.

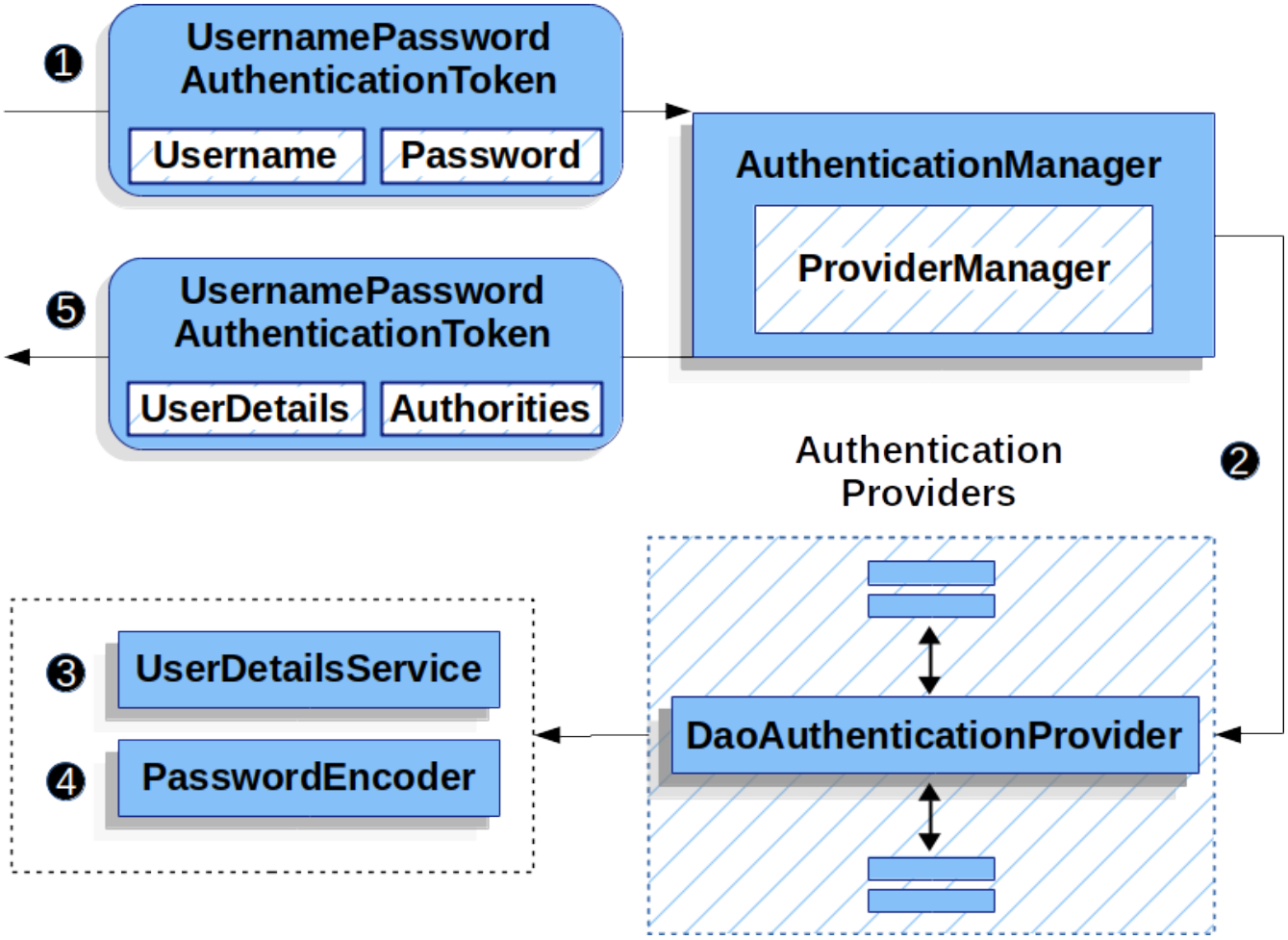


Figure 10. `DaoAuthenticationProvider` *Usage*

1 The authentication `Filter` from Reading the Username & Password passes a `UsernamePasswordAuthenticationToken` to the `AuthenticationManager` which is implemented by `ProviderManager`.

2 The `ProviderManager` is configured to use an AuthenticationProvider of type `DaoAuthenticationProvider`.

3 `DaoAuthenticationProvider` looks up the `UserDetails` from the `UserDetailsService`.

4 `DaoAuthenticationProvider` then uses the `PasswordEncoder` to validate the password on the `UserDetails` returned in the previous step.

5 When authentication is successful, the `Authentication` that is returned is of type `UsernamePasswordAuthenticationToken` and has a principal that is the `UserDetails` returned by the configured `UserDetailsService`. Ultimately, the returned `UsernamePasswordAuthenticationToken` will be set on the `SecurityContextHolder` by the authentication `Filter`.

10.10.10. LDAP Authentication

LDAP is often used by organizations as a central repository for user information and as an authentication service. It can also be used to store the role information for application users.

Spring Security's LDAP based authentication is used by Spring Security when it is configured to accept a username/password for authentication. However, despite leveraging a

username/password for authentication it does not integrate using `UserDetailsService` because in bind authentication the LDAP server does not return the password so the application cannot perform validation of the password.

There are many different scenarios for how an LDAP server may be configured so Spring Security's LDAP provider is fully configurable. It uses separate strategy interfaces for authentication and role retrieval and provides default implementations which can be configured to handle a wide range of situations.

Prerequisites

You should be familiar with LDAP before trying to use it with Spring Security. The following link provides a good introduction to the concepts involved and a guide to setting up a directory using the free LDAP server OpenLDAP: https://www.zytrax.com/books/ldap/. Some familiarity with the JNDI APIs used to access LDAP from Java may also be useful. We don't use any third-party LDAP libraries (Mozilla, JLDAP etc.) in the LDAP provider, but extensive use is made of Spring LDAP, so some familiarity with that project may be useful if you plan on adding your own customizations.

When using LDAP authentication, it is important to ensure that you configure LDAP connection pooling properly. If you are unfamiliar with how to do this, you can refer to the Java LDAP documentation.

Setting up an Embedded LDAP Server

The first thing you will need to do is to ensure that you have an LDAP Server to point your configuration to. For simplicity, it often best to start with an embedded LDAP Server. Spring Security supports using either:

- Embedded UnboundID Server
- Embedded ApacheDS Server

In the samples below, we expose the following as `users.ldif` as a classpath resource to initialize the embedded LDAP server with the users `user` and `admin` both of which have a password of `password`.

users.ldif

```
dn: ou=groups,dc=springframework,dc=org
objectclass: top
objectclass: organizationalUnit
ou: groups

dn: ou=people,dc=springframework,dc=org
objectclass: top
objectclass: organizationalUnit
ou: people

dn: uid=admin,ou=people,dc=springframework,dc=org
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Rod Johnson
sn: Johnson
uid: admin
userPassword: password

dn: uid=user,ou=people,dc=springframework,dc=org
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Dianne Emu
sn: Emu
uid: user
userPassword: password

dn: cn=user,ou=groups,dc=springframework,dc=org
objectclass: top
objectclass: groupOfNames
cn: user
uniqueMember: uid=admin,ou=people,dc=springframework,dc=org
uniqueMember: uid=user,ou=people,dc=springframework,dc=org

dn: cn=admin,ou=groups,dc=springframework,dc=org
objectclass: top
objectclass: groupOfNames
cn: admin
uniqueMember: uid=admin,ou=people,dc=springframework,dc=org
```

Embedded UnboundID Server

If you wish to use UnboundID, then specify the following dependencies:

Example 74. UnboundID Dependencies

Maven

```
<dependency>
    <groupId>com.unboundid</groupId>
    <artifactId>unboundid-ldapsdk</artifactId>
    <version>4.0.14</version>
    <scope>runtime</scope>
</dependency>
```

Gradle

```
depenendencies {
    runtimeOnly "com.unboundid:unboundid-ldapsdk:4.0.14"
}
```

You can then configure the Embedded LDAP Server

Example 75. Embedded LDAP Server Configuration

Java

```
@Bean
UnboundIdContainer ldapContainer() {
    return new UnboundIdContainer("dc=springframework,dc=org",
                "classpath:users.ldif");
}
```

XML

```
<b:bean class="org.springframework.security.ldap.server.UnboundIdContainer"
    c:defaultPartitionSuffix="dc=springframework,dc=org"
    c:ldif="classpath:users.ldif"/>
```

Kotlin

```
@Bean
fun ldapContainer(): UnboundIdContainer {
    return UnboundIdContainer("dc=springframework,dc=org","classpath:users.ldif")
}
```

Embedded ApacheDS Server

Spring Security uses ApacheDS 1.x which is no longer maintained. Unfortunately, ApacheDS 2.x has only released milestone versions with no stable release. Once a stable release of ApacheDS 2.x is available, we will consider updating.

If you wish to use Apache DS, then specify the following dependencies:

Example 76. ApacheDS Dependencies

Maven

```
<dependency>
    <groupId>org.apache.directory.server</groupId>
    <artifactId>apacheds-core</artifactId>
    <version>1.5.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.apache.directory.server</groupId>
    <artifactId>apacheds-server-jndi</artifactId>
    <version>1.5.5</version>
    <scope>runtime</scope>
</dependency>
```

Gradle

```
depenendencies {
    runtimeOnly "org.apache.directory.server:apacheds-core:1.5.5"
    runtimeOnly "org.apache.directory.server:apacheds-server-jndi:1.5.5"
}
```

You can then configure the Embedded LDAP Server

Example 77. Embedded LDAP Server Configuration

Java

```
@Bean
ApacheDSContainer ldapContainer() {
    return new ApacheDSContainer("dc=springframework,dc=org",
                "classpath:users.ldif");
}
```

XML

```
<b:bean class="org.springframework.security.ldap.server.ApacheDSContainer"
    c:defaultPartitionSuffix="dc=springframework,dc=org"
    c:ldif="classpath:users.ldif"/>
```

Kotlin

```
@Bean
fun ldapContainer(): ApacheDSContainer {
    return ApacheDSContainer("dc=springframework,dc=org", "classpath:users.ldif")
}
```

LDAP ContextSource

Once you have an LDAP Server to point your configuration to, you need configure Spring Security to point to an LDAP server that should be used to authenticate users. This is done by creating an LDAP `ContextSource`, which is the equivalent of a JDBC `DataSource`.

Example 78. LDAP Context Source

Java

```
ContextSource contextSource(UnboundIdContainer container) {
    return new
DefaultSpringSecurityContextSource("ldap://localhost:53389/dc=springframework,dc=o
rg");
}
```

XML

```
<ldap-server
    url="ldap://localhost:53389/dc=springframework,dc=org" />
```

Kotlin

```
fun contextSource(container: UnboundIdContainer): ContextSource {
    return
DefaultSpringSecurityContextSource("ldap://localhost:53389/dc=springframework,dc=o
rg")
}
```

Authentication

Spring Security's LDAP support does not use the UserDetailsService because LDAP bind authentication does not allow clients to read the password or even a hashed version of the password. This means there is no way a password to be read and then authenticated by Spring Security.

For this reason, LDAP support is implemented using the `LdapAuthenticator` interface. The `LdapAuthenticator` is also responsible for retrieving any required user attributes. This is because the permissions on the attributes may depend on the type of authentication being used. For example, if binding as the user, it may be necessary to read them with the user's own permissions.

There are two `LdapAuthenticator` implementations supplied with Spring Security:

- Using Bind Authentication
- Using Password Authentication

Using Bind Authentication

Bind Authentication is the most common mechanism for authenticating users with LDAP. In bind authentication the users credentials (i.e. username/password) are submitted to the LDAP server which authenticates them. The advantage to using bind authentication is that the user's secrets (i.e. password) do not need to be exposed to clients which helps to protect them from leaking.

An example of bind authentication configuration can be found below.

Example 79. Bind Authentication

Java

```
@Bean
BindAuthenticator authenticator(BaseLdapPathContextSource contextSource) {
    BindAuthenticator authenticator = new BindAuthenticator(contextSource);
    authenticator.setUserDnPatterns(new String[] { "uid={0},ou=people" });
    return authenticator;
}

@Bean
LdapAuthenticationProvider authenticationProvider(LdapAuthenticator authenticator)
{
    return new LdapAuthenticationProvider(authenticator);
}
```

XML

```
<ldap-authentication-provider
    user-dn-pattern="uid={0},ou=people"/>
```

Kotlin

```
@Bean
fun authenticator(contextSource: BaseLdapPathContextSource): BindAuthenticator {
    val authenticator = BindAuthenticator(contextSource)
    authenticator.setUserDnPatterns(arrayOf("uid={0},ou=people"))
    return authenticator
}

@Bean
fun authenticationProvider(authenticator: LdapAuthenticator):
LdapAuthenticationProvider {
    return LdapAuthenticationProvider(authenticator)
}
```

This simple example would obtain the DN for the user by substituting the user login name in the supplied pattern and attempting to bind as that user with the login password. This is OK if all your users are stored under a single node in the directory. If instead you wished to configure an LDAP search filter to locate the user, you could use the following:

Example 80. Bind Authentication with Search Filter

Java

```
@Bean
BindAuthenticator authenticator(BaseLdapPathContextSource contextSource) {
    String searchBase = "ou=people";
    String filter = "(uid={0})";
    FilterBasedLdapUserSearch search =
        new FilterBasedLdapUserSearch(searchBase, filter, contextSource);
    BindAuthenticator authenticator = new BindAuthenticator(contextSource);
    authenticator.setUserSearch(search);
    return authenticator;
}

@Bean
LdapAuthenticationProvider authenticationProvider(LdapAuthenticator authenticator)
{
    return new LdapAuthenticationProvider(authenticator);
}
```

XML

```
<ldap-authentication-provider
        user-search-filter="(uid={0})"
    user-search-base="ou=people"/>
```

Kotlin

```
@Bean
fun authenticator(contextSource: BaseLdapPathContextSource): BindAuthenticator {
    val searchBase = "ou=people"
    val filter = "(uid={0})"
    val search = FilterBasedLdapUserSearch(searchBase, filter, contextSource)
    val authenticator = BindAuthenticator(contextSource)
    authenticator.setUserSearch(search)
    return authenticator
}

@Bean
fun authenticationProvider(authenticator: LdapAuthenticator):
LdapAuthenticationProvider {
    return LdapAuthenticationProvider(authenticator)
}
```

If used with the `ContextSource` definition above, this would perform a search under the DN `ou=people,dc=springframework,dc=org` using `(uid={0})` as a filter. Again the user login name is substituted for the parameter in the filter name, so it will search for an entry with the `uid` attribute

equal to the user name. If a user search base isn't supplied, the search will be performed from the root.

Using Password Authentication

Password comparison is when the password supplied by the user is compared with the one stored in the repository. This can either be done by retrieving the value of the password attribute and checking it locally or by performing an LDAP "compare" operation, where the supplied password is passed to the server for comparison and the real password value is never retrieved. An LDAP compare cannot be done when the password is properly hashed with a random salt.

Example 81. Minimal Password Compare Configuration

Java

```
@Bean
PasswordComparisonAuthenticator authenticator(BaseLdapPathContextSource
contextSource) {
    return new PasswordComparisonAuthenticator(contextSource);
}

@Bean
LdapAuthenticationProvider authenticationProvider(LdapAuthenticator authenticator)
{
    return new LdapAuthenticationProvider(authenticator);
}
```

XML

```
<ldap-authentication-provider
        user-dn-pattern="uid={0},ou=people">
    <password-compare />
</ldap-authentication-provider>
```

Kotlin

```
@Bean
fun authenticator(contextSource: BaseLdapPathContextSource):
PasswordComparisonAuthenticator {
    return PasswordComparisonAuthenticator(contextSource)
}

@Bean
fun authenticationProvider(authenticator: LdapAuthenticator):
LdapAuthenticationProvider {
    return LdapAuthenticationProvider(authenticator)
}
```

A more advanced configuration with some customizations can be found below.

Example 82. Password Compare Configuration

Java

```
@Bean
PasswordComparisonAuthenticator authenticator(BaseLdapPathContextSource
contextSource) {
    PasswordComparisonAuthenticator authenticator =
        new PasswordComparisonAuthenticator(contextSource);
    authenticator.setPasswordAttributeName("pwd"); ①
    authenticator.setPasswordEncoder(new BCryptPasswordEncoder()); ②
    return authenticator;
}

@Bean
LdapAuthenticationProvider authenticationProvider(LdapAuthenticator authenticator)
{
    return new LdapAuthenticationProvider(authenticator);
}
```

XML

```
<ldap-authentication-provider
        user-dn-pattern="uid={0},ou=people">
    <password-compare password-attribute="pwd"> ①
        <password-encoder ref="passwordEncoder" /> ②
    </password-compare>
</ldap-authentication-provider>
<b:bean id="passwordEncoder"
    class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder" />
```

Kotlin

```
@Bean
fun authenticator(contextSource: BaseLdapPathContextSource):
PasswordComparisonAuthenticator {
    val authenticator = PasswordComparisonAuthenticator(contextSource)
    authenticator.setPasswordAttributeName("pwd") ①
    authenticator.setPasswordEncoder(BCryptPasswordEncoder()) ②
    return authenticator
}

@Bean
fun authenticationProvider(authenticator: LdapAuthenticator):
LdapAuthenticationProvider {
    return LdapAuthenticationProvider(authenticator)
}
```

① Specify the password attribute as `pwd`

② Use `BCryptPasswordEncoder`

LdapAuthoritiesPopulator

Spring Security's `LdapAuthoritiesPopulator` is used to determine what authorites are returned for the user.

Example 83. LdapAuthoritiesPopulator Configuration

Java

```
@Bean
LdapAuthoritiesPopulator authorities(BaseLdapPathContextSource contextSource) {
    String groupSearchBase = "";
    DefaultLdapAuthoritiesPopulator authorities =
        new DefaultLdapAuthoritiesPopulator(contextSource, groupSearchBase);
    authorities.setGroupSearchFilter("member={0}");
    return authorities;
}

@Bean
LdapAuthenticationProvider authenticationProvider(LdapAuthenticator authenticator,
LdapAuthoritiesPopulator authorities) {
    return new LdapAuthenticationProvider(authenticator, authorities);
}
```

XML

```
<ldap-authentication-provider
    user-dn-pattern="uid={0},ou=people"
    group-search-filter="member={0}"/>
```

Kotlin

```
@Bean
fun authorities(contextSource: BaseLdapPathContextSource):
LdapAuthoritiesPopulator {
    val groupSearchBase = ""
    val authorities = DefaultLdapAuthoritiesPopulator(contextSource,
groupSearchBase)
    authorities.setGroupSearchFilter("member={0}")
    return authorities
}

@Bean
fun authenticationProvider(authenticator: LdapAuthenticator, authorities:
LdapAuthoritiesPopulator): LdapAuthenticationProvider {
    return LdapAuthenticationProvider(authenticator, authorities)
}
```

Active Directory

Active Directory supports its own non-standard authentication options, and the normal usage pattern doesn't fit too cleanly with the standard `LdapAuthenticationProvider`. Typically authentication is performed using the domain username (in the form `user@domain`), rather than

using an LDAP distinguished name. To make this easier, Spring Security has an authentication provider which is customized for a typical Active Directory setup.

Configuring `ActiveDirectoryLdapAuthenticationProvider` is quite straightforward. You just need to supply the domain name and an LDAP URL supplying the address of the server [2]. An example configuration can be seen below:

Example 84. Example Active Directory Configuration

Java

```
@Bean
ActiveDirectoryLdapAuthenticationProvider authenticationProvider() {
    return new ActiveDirectoryLdapAuthenticationProvider("example.com",
"ldap://company.example.com/");
}
```

XML

```
<bean id="authenticationProvider"

class="org.springframework.security.ldap.authentication.ad.ActiveDirectoryLdapAuth
enticationProvider">
    <constructor-arg value="example.com" />
    <constructor-arg value="ldap://company.example.com/" />
</bean>
```

Kotlin

```
@Bean
fun authenticationProvider(): ActiveDirectoryLdapAuthenticationProvider {
    return ActiveDirectoryLdapAuthenticationProvider("example.com",
"ldap://company.example.com/")
}
```

10.11. Session Management

HTTP session related functionality is handled by a combination of the `SessionManagementFilter` and the `SessionAuthenticationStrategy` interface, which the filter delegates to. Typical usage includes session-fixation protection attack prevention, detection of session timeouts and restrictions on how many sessions an authenticated user may have open concurrently.

10.11.1. Detecting Timeouts

You can configure Spring Security to detect the submission of an invalid session ID and redirect the user to an appropriate URL. This is achieved through the `session-management` element:

```
<http>
...
<session-management invalid-session-url="/invalidSession.htm" />
</http>
```

Note that if you use this mechanism to detect session timeouts, it may falsely report an error if the user logs out and then logs back in without closing the browser. This is because the session cookie is not cleared when you invalidate the session and will be resubmitted even if the user has logged out. You may be able to explicitly delete the JSESSIONID cookie on logging out, for example by using the following syntax in the logout handler:

```
<http>
<logout delete-cookies="JSESSIONID" />
</http>
```

Unfortunately this can't be guaranteed to work with every servlet container, so you will need to test it in your environment

If you are running your application behind a proxy, you may also be able to remove the session cookie by configuring the proxy server. For example, using Apache HTTPD's mod_headers, the following directive would delete the `JSESSIONID` cookie by expiring it in the response to a logout request (assuming the application is deployed under the path `/tutorial`):

```
<LocationMatch "/tutorial/logout">
Header always set Set-Cookie "JSESSIONID=;Path=/tutorial;Expires=Thu,
01 Jan 1970 00:00:00 GMT"
</LocationMatch>
```

10.11.2. Concurrent Session Control

If you wish to place constraints on a single user's ability to log in to your application, Spring Security supports this out of the box with the following simple additions. First you need to add the following listener to your `web.xml` file to keep Spring Security updated about session lifecycle events:

```
<listener>
<listener-class>
    org.springframework.security.web.session.HttpSessionEventPublisher
</listener-class>
</listener>
```

Then add the following lines to your application context:

```
<http>
...
<session-management>
    <concurrency-control max-sessions="1" />
</session-management>
</http>
```

This will prevent a user from logging in multiple times - a second login will cause the first to be invalidated. Often you would prefer to prevent a second login, in which case you can use

```
<http>
...
<session-management>
    <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" />
</session-management>
</http>
```

The second login will then be rejected. By "rejected", we mean that the user will be sent to the `authentication-failure-url` if form-based login is being used. If the second authentication takes place through another non-interactive mechanism, such as "remember-me", an "unauthorized" (401) error will be sent to the client. If instead you want to use an error page, you can add the attribute `session-authentication-error-url` to the `session-management` element.

If you are using a customized authentication filter for form-based login, then you have to configure concurrent session control support explicitly. More details can be found in the Session Management chapter.

10.11.3. Session Fixation Attack Protection

Session fixation attacks are a potential risk where it is possible for a malicious attacker to create a session by accessing a site, then persuade another user to log in with the same session (by sending them a link containing the session identifier as a parameter, for example). Spring Security protects against this automatically by creating a new session or otherwise changing the session ID when a user logs in. If you don't require this protection, or it conflicts with some other requirement, you can control the behavior using the `session-fixation-protection` attribute on `<session-management>`, which has four options

- `none` - Don't do anything. The original session will be retained.
- `newSession` - Create a new "clean" session, without copying the existing session data (Spring Security-related attributes will still be copied).
- `migrateSession` - Create a new session and copy all existing session attributes to the new session. This is the default in Servlet 3.0 or older containers.
- `changeSessionId` - Do not create a new session. Instead, use the session fixation protection provided by the Servlet container (`HttpServletRequest#changeSessionId()`). This option is only available in Servlet 3.1 (Java EE 7) and newer containers. Specifying it in older containers will result in an exception. This is the default in Servlet 3.1 and newer containers.

When session fixation protection occurs, it results in a `SessionFixationProtectionEvent` being published in the application context. If you use `changeSessionId`, this protection will *also* result in any `javax.servlet.http.HttpSessionIdListener` s being notified, so use caution if your code listens for both events. See the Session Management chapter for additional information.

10.11.4. SessionManagementFilter

The `SessionManagementFilter` checks the contents of the `SecurityContextRepository` against the current contents of the `SecurityContextHolder` to determine whether a user has been authenticated during the current request, typically by a non-interactive authentication mechanism, such as pre-authentication or remember-me [3]. If the repository contains a security context, the filter does nothing. If it doesn't, and the thread-local `SecurityContext` contains a (non-anonymous) `Authentication` object, the filter assumes they have been authenticated by a previous filter in the stack. It will then invoke the configured `SessionAuthenticationStrategy`.

If the user is not currently authenticated, the filter will check whether an invalid session ID has been requested (because of a timeout, for example) and will invoke the configured `InvalidSessionStrategy`, if one is set. The most common behaviour is just to redirect to a fixed URL and this is encapsulated in the standard implementation `SimpleRedirectInvalidSessionStrategy`. The latter is also used when configuring an invalid session URL through the namespace,as described earlier.

10.11.5. SessionAuthenticationStrategy

`SessionAuthenticationStrategy` is used by both `SessionManagementFilter` and `AbstractAuthenticationProcessingFilter`, so if you are using a customized form-login class, for example, you will need to inject it into both of these. In this case, a typical configuration, combining the namespace and custom beans might look like this:

```
<http>
<custom-filter position="FORM_LOGIN_FILTER" ref="myAuthFilter" />
<session-management session-authentication-strategy-ref="sas"/>
</http>

<beans:bean id="myAuthFilter" class=
"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter"
>
    <beans:property name="sessionAuthenticationStrategy" ref="sas" />
    ...
</beans:bean>

<beans:bean id="sas" class=
"org.springframework.security.web.authentication.session.SessionFixationProtectionStra
tegy" />
```

Note that the use of the default, `SessionFixationProtectionStrategy` may cause issues if you are storing beans in the session which implement `HttpSessionBindingListener`, including Spring session-scoped beans. See the Javadoc for this class for more information.

10.11.6. Concurrency Control

Spring Security is able to prevent a principal from concurrently authenticating to the same application more than a specified number of times. Many ISVs take advantage of this to enforce licensing, whilst network administrators like this feature because it helps prevent people from sharing login names. You can, for example, stop user "Batman" from logging onto the web application from two different sessions. You can either expire their previous login or you can report an error when they try to log in again, preventing the second login. Note that if you are using the second approach, a user who has not explicitly logged out (but who has just closed their browser, for example) will not be able to log in again until their original session expires.

Concurrency control is supported by the namespace, so please check the earlier namespace chapter for the simplest configuration. Sometimes you need to customize things though.

The implementation uses a specialized version of `SessionAuthenticationStrategy`, called `ConcurrentSessionControlAuthenticationStrategy`.

Previously the concurrent authentication check was made by the `ProviderManager`, which could be injected with a `ConcurrentSessionController`. The latter would check if the user was attempting to exceed the number of permitted sessions. However, this approach required that an HTTP session be created in advance, which is undesirable. In Spring Security 3, the user is first authenticated by the `AuthenticationManager` and once they are successfully authenticated, a session is created and the check is made whether they are allowed to have another session open.

To use concurrent session support, you'll need to add the following to `web.xml`:

```
<listener>
    <listener-class>
    org.springframework.security.web.session.HttpSessionEventPublisher
    </listener-class>
</listener>
```

In addition, you will need to add the `ConcurrentSessionFilter` to your `FilterChainProxy`. The `ConcurrentSessionFilter` requires two constructor arguments, `sessionRegistry`, which generally points to an instance of `SessionRegistryImpl`, and `sessionInformationExpiredStrategy`, which defines the strategy to apply when a session has expired. A configuration using the namespace to create the `FilterChainProxy` and other default beans might look like this:

```
<http>
<custom-filter position="CONCURRENT_SESSION_FILTER" ref="concurrencyFilter" />
<custom-filter position="FORM_LOGIN_FILTER" ref="myAuthFilter" />

<session-management session-authentication-strategy-ref="sas"/>
</http>

<beans:bean id="redirectSessionInformationExpiredStrategy"
```

```
class="org.springframework.security.web.session.SimpleRedirectSessionInformationExpire
dStrategy">
<beans:constructor-arg name="invalidSessionUrl" value="/session-expired.htm" />
</beans:bean>

<beans:bean id="concurrencyFilter"
class="org.springframework.security.web.session.ConcurrentSessionFilter">
<beans:constructor-arg name="sessionRegistry" ref="sessionRegistry" />
<beans:constructor-arg name="sessionInformationExpiredStrategy"
ref="redirectSessionInformationExpiredStrategy" />
</beans:bean>

<beans:bean id="myAuthFilter" class=
"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter"
>
<beans:property name="sessionAuthenticationStrategy" ref="sas" />
<beans:property name="authenticationManager" ref="authenticationManager" />
</beans:bean>

<beans:bean id="sas"
class="org.springframework.security.web.authentication.session.CompositeSessionAuthent
icationStrategy">
<beans:constructor-arg>
    <beans:list>
    <beans:bean
class="org.springframework.security.web.authentication.session.ConcurrentSessionContro
lAuthenticationStrategy">
        <beans:constructor-arg ref="sessionRegistry"/>
        <beans:property name="maximumSessions" value="1" />
        <beans:property name="exceptionIfMaximumExceeded" value="true" />
    </beans:bean>
    <beans:bean
class="org.springframework.security.web.authentication.session.SessionFixationProtecti
onStrategy">
    </beans:bean>
    <beans:bean
class="org.springframework.security.web.authentication.session.RegisterSessionAuthenti
cationStrategy">
        <beans:constructor-arg ref="sessionRegistry"/>
    </beans:bean>
    </beans:list>
</beans:constructor-arg>
</beans:bean>

<beans:bean id="sessionRegistry"
    class="org.springframework.security.core.session.SessionRegistryImpl" />
```

Adding the listener to `web.xml` causes an `ApplicationEvent` to be published to the Spring `ApplicationContext` every time a `HttpSession` commences or ends. This is critical, as it allows the `SessionRegistryImpl` to be notified when a session ends. Without it, a user will never be able to log

back in again once they have exceeded their session allowance, even if they log out of another session or it times out.

Querying the SessionRegistry for currently authenticated users and their sessions

Setting up concurrency-control, either through the namespace or using plain beans has the useful side effect of providing you with a reference to the `SessionRegistry` which you can use directly within your application, so even if you don't want to restrict the number of sessions a user may have, it may be worth setting up the infrastructure anyway. You can set the `maximumSession` property to -1 to allow unlimited sessions. If you're using the namespace, you can set an alias for the internally-created `SessionRegistry` using the `session-registry-alias` attribute, providing a reference which you can inject into your own beans.

The `getAllPrincipals()` method supplies you with a list of the currently authenticated users. You can list a user's sessions by calling the `getAllSessions(Object principal, boolean includeExpiredSessions)` method, which returns a list of `SessionInformation` objects. You can also expire a user's session by calling `expireNow()` on a `SessionInformation` instance. When the user returns to the application, they will be prevented from proceeding. You may find these methods useful in an administration application, for example. Have a look at the Javadoc for more information.

10.12. Remember-Me Authentication

10.12.1. Overview

Remember-me or persistent-login authentication refers to web sites being able to remember the identity of a principal between sessions. This is typically accomplished by sending a cookie to the browser, with the cookie being detected during future sessions and causing automated login to take place. Spring Security provides the necessary hooks for these operations to take place, and has two concrete remember-me implementations. One uses hashing to preserve the security of cookie-based tokens and the other uses a database or other persistent storage mechanism to store the generated tokens.

Note that both implementations require a `UserDetailsService`. If you are using an authentication provider which doesn't use a `UserDetailsService` (for example, the LDAP provider) then it won't work unless you also have a `UserDetailsService` bean in your application context.

10.12.2. Simple Hash-Based Token Approach

This approach uses hashing to achieve a useful remember-me strategy. In essence a cookie is sent to the browser upon successful interactive authentication, with the cookie being composed as follows:

```
base64(username + ":" + expirationTime + ":" +
md5Hex(username + ":" + expirationTime + ":" password + ":" + key))

username:          As identifiable to the UserDetailsService
password:          That matches the one in the retrieved UserDetails
expirationTime:    The date and time when the remember-me token expires, expressed in
milliseconds
key:               A private key to prevent modification of the remember-me token
```

As such the remember-me token is valid only for the period specified, and provided that the username, password and key does not change. Notably, this has a potential security issue in that a captured remember-me token will be usable from any user agent until such time as the token expires. This is the same issue as with digest authentication. If a principal is aware a token has been captured, they can easily change their password and immediately invalidate all remember-me tokens on issue. If more significant security is needed you should use the approach described in the next section. Alternatively remember-me services should simply not be used at all.

If you are familiar with the topics discussed in the chapter on namespace configuration, you can enable remember-me authentication just by adding the `<remember-me>` element:

```
<http>
...
<remember-me key="myAppKey"/>
</http>
```

The `UserDetailsService` will normally be selected automatically. If you have more than one in your application context, you need to specify which one should be used with the `user-service-ref` attribute, where the value is the name of your `UserDetailsService` bean.

10.12.3. Persistent Token Approach

This approach is based on the article http://jaspan.com/improved_persistent_login_cookie_best_practice with some minor modifications [4]. To use the this approach with namespace configuration, you would supply a datasource reference:

```
<http>
...
<remember-me data-source-ref="someDataSource"/>
</http>
```

The database should contain a `persistent_logins` table, created using the following SQL (or equivalent):

```
create table persistent_logins (username varchar(64) not null,
                                series varchar(64) primary key,
                                token varchar(64) not null,
                                last_used timestamp not null)
```

10.12.4. Remember-Me Interfaces and Implementations

Remember-me is used with `UsernamePasswordAuthenticationFilter`, and is implemented via hooks in the `AbstractAuthenticationProcessingFilter` superclass. It is also used within `BasicAuthenticationFilter`. The hooks will invoke a concrete `RememberMeServices` at the appropriate times. The interface looks like this:

```
Authentication autoLogin(HttpServletRequest request, HttpServletResponse response);

void loginFail(HttpServletRequest request, HttpServletResponse response);

void loginSuccess(HttpServletRequest request, HttpServletResponse response,
    Authentication successfulAuthentication);
```

Please refer to the Javadoc for a fuller discussion on what the methods do, although note at this stage that `AbstractAuthenticationProcessingFilter` only calls the `loginFail()` and `loginSuccess()` methods. The `autoLogin()` method is called by `RememberMeAuthenticationFilter` whenever the `SecurityContextHolder` does not contain an `Authentication`. This interface therefore provides the underlying remember-me implementation with sufficient notification of authentication-related events, and delegates to the implementation whenever a candidate web request might contain a cookie and wish to be remembered. This design allows any number of remember-me implementation strategies. We've seen above that Spring Security provides two implementations. We'll look at these in turn.

TokenBasedRememberMeServices

This implementation supports the simpler approach described in Simple Hash-Based Token Approach. `TokenBasedRememberMeServices` generates a `RememberMeAuthenticationToken`, which is processed by `RememberMeAuthenticationProvider`. A `key` is shared between this authentication provider and the `TokenBasedRememberMeServices`. In addition, `TokenBasedRememberMeServices` requires A UserDetailsService from which it can retrieve the username and password for signature comparison purposes, and generate the `RememberMeAuthenticationToken` to contain the correct `GrantedAuthority` s. Some sort of logout command should be provided by the application that invalidates the cookie if the user requests this. `TokenBasedRememberMeServices` also implements Spring Security's `LogoutHandler` interface so can be used with `LogoutFilter` to have the cookie cleared automatically.

The beans required in an application context to enable remember-me services are as follows:

```
<bean id="rememberMeFilter" class=
"org.springframework.security.web.authentication.rememberme.RememberMeAuthenticationFi
lter">
<property name="rememberMeServices" ref="rememberMeServices"/>
<property name="authenticationManager" ref="theAuthenticationManager" />
</bean>

<bean id="rememberMeServices" class=
"org.springframework.security.web.authentication.rememberme.TokenBasedRememberMeServic
es">
<property name="userDetailsService" ref="myUserDetailsService"/>
<property name="key" value="springRocks"/>
</bean>

<bean id="rememberMeAuthenticationProvider" class=
"org.springframework.security.authentication.RememberMeAuthenticationProvider">
<property name="key" value="springRocks"/>
</bean>
```

Don't forget to add your `RememberMeServices` implementation to your `UsernamePasswordAuthenticationFilter.setRememberMeServices()` property, include the `RememberMeAuthenticationProvider` in your `AuthenticationManager.setProviders()` list, and add `RememberMeAuthenticationFilter` into your `FilterChainProxy` (typically immediately after your `UsernamePasswordAuthenticationFilter`).

PersistentTokenBasedRememberMeServices

This class can be used in the same way as `TokenBasedRememberMeServices`, but it additionally needs to be configured with a `PersistentTokenRepository` to store the tokens. There are two standard implementations.

- `InMemoryTokenRepositoryImpl` which is intended for testing only.
- `JdbcTokenRepositoryImpl` which stores the tokens in a database.

The database schema is described above in Persistent Token Approach.

10.13. OpenID Support

The OpenID 1.0 and 2.0 protocols have been deprecated and users are encouraged to migrate to OpenID Connect, which is supported by spring-security-oauth2.

The namespace supports OpenID login either instead of, or in addition to normal form-based login, with a simple change:

```
<http>
<intercept-url pattern="/**" access="ROLE_USER" />
<openid-login />
</http>
```

You should then register yourself with an OpenID provider (such as myopenid.com), and add the user information to your in-memory `<user-service>`:

```
<user name="https://jimi.hendrix.myopenid.com/" authorities="ROLE_USER" />
```

You should be able to login using the `myopenid.com` site to authenticate. It is also possible to select a specific `UserDetailsService` bean for use OpenID by setting the `user-service-ref` attribute on the `openid-login` element. Note that we have omitted the password attribute from the above user configuration, since this set of user data is only being used to load the authorities for the user. A random password will be generated internally, preventing you from accidentally using this user data as an authentication source elsewhere in your configuration.

10.13.1. Attribute Exchange

Support for OpenID attribute exchange. As an example, the following configuration would attempt to retrieve the email and full name from the OpenID provider, for use by the application:

```
<openid-login>
<attribute-exchange>
    <openid-attribute name="email" type="https://axschema.org/contact/email"
required="true"/>
    <openid-attribute name="name" type="https://axschema.org/namePerson"/>
</attribute-exchange>
</openid-login>
```

The "type" of each OpenID attribute is a URI, determined by a particular schema, in this case https://axschema.org/. If an attribute must be retrieved for successful authentication, the `required` attribute can be set. The exact schema and attributes supported will depend on your OpenID provider. The attribute values are returned as part of the authentication process and can be accessed afterwards using the following code:

```
OpenIDAuthenticationToken token =
    (OpenIDAuthenticationToken)SecurityContextHolder.getContext().getAuthentication();
List<OpenIDAttribute> attributes = token.getAttributes();
```

We can obtain the `OpenIDAuthenticationToken` from the SecurityContextHolder. The `OpenIDAttribute` contains the attribute type and the retrieved value (or values in the case of multi-valued attributes). You can supply multiple `attribute-exchange` elements, using an `identifier-matcher` attribute on each. This contains a regular expression which will be matched against the OpenID identifier supplied by the user. See the OpenID sample application in the codebase for an example

configuration, providing different attribute lists for the Google, Yahoo and MyOpenID providers.

10.14. Anonymous Authentication

10.14.1. Overview

It's generally considered good security practice to adopt a "deny-by-default" where you explicitly specify what is allowed and disallow everything else. Defining what is accessible to unauthenticated users is a similar situation, particularly for web applications. Many sites require that users must be authenticated for anything other than a few URLs (for example the home and login pages). In this case it is easiest to define access configuration attributes for these specific URLs rather than have for every secured resource. Put differently, sometimes it is nice to say `ROLE_SOMETHING` is required by default and only allow certain exceptions to this rule, such as for login, logout and home pages of an application. You could also omit these pages from the filter chain entirely, thus bypassing the access control checks, but this may be undesirable for other reasons, particularly if the pages behave differently for authenticated users.

This is what we mean by anonymous authentication. Note that there is no real conceptual difference between a user who is "anonymously authenticated" and an unauthenticated user. Spring Security's anonymous authentication just gives you a more convenient way to configure your access-control attributes. Calls to servlet API calls such as `getCallerPrincipal`, for example, will still return null even though there is actually an anonymous authentication object in the `SecurityContextHolder`.

There are other situations where anonymous authentication is useful, such as when an auditing interceptor queries the `SecurityContextHolder` to identify which principal was responsible for a given operation. Classes can be authored more robustly if they know the `SecurityContextHolder` always contains an `Authentication` object, and never `null`.

10.14.2. Configuration

Anonymous authentication support is provided automatically when using the HTTP configuration Spring Security 3.0 and can be customized (or disabled) using the `<anonymous>` element. You don't need to configure the beans described here unless you are using traditional bean configuration.

Three classes that together provide the anonymous authentication feature. `AnonymousAuthenticationToken` is an implementation of `Authentication`, and stores the `GrantedAuthority` s which apply to the anonymous principal. There is a corresponding `AnonymousAuthenticationProvider`, which is chained into the `ProviderManager` so that `AnonymousAuthenticationToken` s are accepted. Finally, there is an `AnonymousAuthenticationFilter`, which is chained after the normal authentication mechanisms and automatically adds an `AnonymousAuthenticationToken` to the `SecurityContextHolder` if there is no existing `Authentication` held there. The definition of the filter and authentication provider appears as follows:

```
<bean id="anonymousAuthFilter"

class="org.springframework.security.web.authentication.AnonymousAuthenticationFilter">
<property name="key" value="foobar"/>
<property name="userAttribute" value="anonymousUser,ROLE_ANONYMOUS"/>
</bean>

<bean id="anonymousAuthenticationProvider"

class="org.springframework.security.authentication.AnonymousAuthenticationProvider">
<property name="key" value="foobar"/>
</bean>
```

The `key` is shared between the filter and authentication provider, so that tokens created by the former are accepted by the latter [5]. The `userAttribute` is expressed in the form of `usernameInTheAuthenticationToken,grantedAuthority[,grantedAuthority]`. This is the same syntax as used after the equals sign for the `userMap` property of `InMemoryDaoImpl`.

As explained earlier, the benefit of anonymous authentication is that all URI patterns can have security applied to them. For example:

```
<bean id="filterSecurityInterceptor"

class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="httpRequestAccessDecisionManager"/>
<property name="securityMetadata">
    <security:filter-security-metadata-source>
    <security:intercept-url pattern='/index.jsp' access='ROLE_ANONYMOUS,ROLE_USER'/>
    <security:intercept-url pattern='/hello.htm' access='ROLE_ANONYMOUS,ROLE_USER'/>
    <security:intercept-url pattern='/logoff.jsp' access='ROLE_ANONYMOUS,ROLE_USER'/>
    <security:intercept-url pattern='/login.jsp' access='ROLE_ANONYMOUS,ROLE_USER'/>
    <security:intercept-url pattern='/**' access='ROLE_USER'/>
    </security:filter-security-metadata-source>" +
</property>
</bean>
```

10.14.3. AuthenticationTrustResolver

Rounding out the anonymous authentication discussion is the `AuthenticationTrustResolver` interface, with its corresponding `AuthenticationTrustResolverImpl` implementation. This interface provides an `isAnonymous(Authentication)` method, which allows interested classes to take into account this special type of authentication status. The `ExceptionTranslationFilter` uses this interface in processing `AccessDeniedException` s. If an `AccessDeniedException` is thrown, and the authentication is of an anonymous type, instead of throwing a 403 (forbidden) response, the filter will instead commence the `AuthenticationEntryPoint` so the principal can authenticate properly. This is a necessary distinction, otherwise principals would always be deemed "authenticated" and

never be given an opportunity to login via form, basic, digest or some other normal authentication mechanism.

You will often see the `ROLE_ANONYMOUS` attribute in the above interceptor configuration replaced with `IS_AUTHENTICATED_ANONYMOUSLY`, which is effectively the same thing when defining access controls. This is an example of the use of the `AuthenticatedVoter` which we will see in the authorization chapter. It uses an `AuthenticationTrustResolver` to process this particular configuration attribute and grant access to anonymous users. The `AuthenticatedVoter` approach is more powerful, since it allows you to differentiate between anonymous, remember-me and fully-authenticated users. If you don't need this functionality though, then you can stick with `ROLE_ANONYMOUS`, which will be processed by Spring Security's standard `RoleVoter`.

10.14.4. Getting Anonymous Authentications with Spring MVC

Spring MVC resolves parameters of type `Principal` using its own argument resolver.

This means that a construct like this one:

Java

```
@GetMapping("/")
public String method(Authentication authentication) {
    if (authentication instanceof AnonymousAuthenticationToken) {
        return "anonymous";
    } else {
        return "not anonymous";
    }
}
```

Kotlin

```
@GetMapping("/")
fun method(authentication: Authentication?): String {
    return if (authentication is AnonymousAuthenticationToken) {
        "anonymous"
    } else {
        "not anonymous"
    }
}
```

will always return "not anonymous", even for anonymous requests. The reason is that Spring MVC resolves the parameter using `HttpServletRequest#getPrincipal`, which is `null` when the request is anonymous.

If you'd like to obtain the `Authentication` in anonymous requests, use `@CurrentSecurityContext` instead:

Example 85. Use CurrentSecurityContext for Anonymous requests

Java

```
@GetMapping("/")
public String method(@CurrentSecurityContext SecurityContext context) {
    return context.getAuthentication().getName();
}
```

Kotlin

```
@GetMapping("/")
fun method(@CurrentSecurityContext context : SecurityContext) : String =
        context!!.authentication!!.name
```

10.15. Pre-Authentication Scenarios

There are situations where you want to use Spring Security for authorization, but the user has already been reliably authenticated by some external system prior to accessing the application. We refer to these situations as "pre-authenticated" scenarios. Examples include X.509, Siteminder and authentication by the Java EE container in which the application is running. When using pre-authentication, Spring Security has to

- Identify the user making the request.
- Obtain the authorities for the user.

The details will depend on the external authentication mechanism. A user might be identified by their certificate information in the case of X.509, or by an HTTP request header in the case of Siteminder. If relying on container authentication, the user will be identified by calling the `getUserPrincipal()` method on the incoming HTTP request. In some cases, the external mechanism may supply role/authority information for the user but in others the authorities must be obtained from a separate source, such as a `UserDetailsService`.

10.15.1. Pre-Authentication Framework Classes

Because most pre-authentication mechanisms follow the same pattern, Spring Security has a set of classes which provide an internal framework for implementing pre-authenticated authentication providers. This removes duplication and allows new implementations to be added in a structured fashion, without having to write everything from scratch. You don't need to know about these classes if you want to use something like X.509 authentication, as it already has a namespace configuration option which is simpler to use and get started with. If you need to use explicit bean configuration or are planning on writing your own implementation then an understanding of how the provided implementations work will be useful. You will find classes under the `org.springframework.security.web.authentication.preauth`. We just provide an outline here so you should consult the Javadoc and source where appropriate.

AbstractPreAuthenticatedProcessingFilter

This class will check the current contents of the security context and, if empty, it will attempt to extract user information from the HTTP request and submit it to the `AuthenticationManager`. Subclasses override the following methods to obtain this information:

Example 86. Override AbstractPreAuthenticatedProcessingFilter

Java

```
protected abstract Object getPreAuthenticatedPrincipal(HttpServletRequest
request);

protected abstract Object getPreAuthenticatedCredentials(HttpServletRequest
request);
```

Kotlin

```
protected abstract fun getPreAuthenticatedPrincipal(request: HttpServletRequest):
Any?

protected abstract fun getPreAuthenticatedCredentials(request:
HttpServletRequest): Any?
```

After calling these, the filter will create a `PreAuthenticatedAuthenticationToken` containing the returned data and submit it for authentication. By "authentication" here, we really just mean further processing to perhaps load the user's authorities, but the standard Spring Security authentication architecture is followed.

Like other Spring Security authentication filters, the pre-authentication filter has an `authenticationDetailsSource` property which by default will create a `WebAuthenticationDetails` object to store additional information such as the session-identifier and originating IP address in the `details` property of the `Authentication` object. In cases where user role information can be obtained from the pre-authentication mechanism, the data is also stored in this property, with the details implementing the `GrantedAuthoritiesContainer` interface. This enables the authentication provider to read the authorities which were externally allocated to the user. We'll look at a concrete example next.

J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource

If the filter is configured with an `authenticationDetailsSource` which is an instance of this class, the authority information is obtained by calling the `isUserInRole(String role)` method for each of a pre-determined set of "mappable roles". The class gets these from a configured `MappableAttributesRetriever`. Possible implementations include hard-coding a list in the application context and reading the role information from the `<security-role>` information in a `web.xml` file. The pre-authentication sample application uses the latter approach.

There is an additional stage where the roles (or attributes) are mapped to Spring Security

`GrantedAuthority` objects using a configured `Attributes2GrantedAuthoritiesMapper`. The default will just add the usual `ROLE_` prefix to the names, but it gives you full control over the behaviour.

PreAuthenticatedAuthenticationProvider

The pre-authenticated provider has little more to do than load the `UserDetails` object for the user. It does this by delegating to an `AuthenticationUserDetailsService`. The latter is similar to the standard `UserDetailsService` but takes an `Authentication` object rather than just user name:

```
public interface AuthenticationUserDetailsService {
    UserDetails loadUserDetails(Authentication token) throws
UsernameNotFoundException;
}
```

This interface may have also other uses but with pre-authentication it allows access to the authorities which were packaged in the `Authentication` object, as we saw in the previous section. The `PreAuthenticatedGrantedAuthoritiesUserDetailsService` class does this. Alternatively, it may delegate to a standard `UserDetailsService` via the `UserDetailsByNameServiceWrapper` implementation.

Http403ForbiddenEntryPoint

The `AuthenticationEntryPoint` is responsible for kick-starting the authentication process for an unauthenticated user (when they try to access a protected resource), but in the pre-authenticated case this doesn't apply. You would only configure the `ExceptionTranslationFilter` with an instance of this class if you aren't using pre-authentication in combination with other authentication mechanisms. It will be called if the user is rejected by the `AbstractPreAuthenticatedProcessingFilter` resulting in a null authentication. It always returns a `403`-forbidden response code if called.

10.15.2. Concrete Implementations

X.509 authentication is covered in its own chapter. Here we'll look at some classes which provide support for other pre-authenticated scenarios.

Request-Header Authentication (Siteminder)

An external authentication system may supply information to the application by setting specific headers on the HTTP request. A well-known example of this is Siteminder, which passes the username in a header called `SM_USER`. This mechanism is supported by the class `RequestHeaderAuthenticationFilter` which simply extracts the username from the header. It defaults to using the name `SM_USER` as the header name. See the Javadoc for more details.

Note that when using a system like this, the framework performs no authentication checks at all and it is *extremely* important that the external system is configured properly and protects all access to the application. If an attacker is able to forge the headers in their original request without this being detected then they could potentially choose any username they wished.

Siteminder Example Configuration

A typical configuration using this filter would look like this:

```
<security:http>
<!-- Additional http configuration omitted -->
<security:custom-filter position="PRE_AUTH_FILTER" ref="siteminderFilter" />
</security:http>

<bean id="siteminderFilter"
class="org.springframework.security.web.authentication.preauth.RequestHeaderAuthentica
tionFilter">
<property name="principalRequestHeader" value="SM_USER"/>
<property name="authenticationManager" ref="authenticationManager" />
</bean>

<bean id="preauthAuthProvider"
class="org.springframework.security.web.authentication.preauth.PreAuthenticatedAuthent
icationProvider">
<property name="preAuthenticatedUserDetailsService">
    <bean id="userDetailsServiceWrapper"

class="org.springframework.security.core.userdetails.UserDetailsByNameServiceWrapper">
    <property name="userDetailsService" ref="userDetailsService"/>
    </bean>
</property>
</bean>

<security:authentication-manager alias="authenticationManager">
<security:authentication-provider ref="preauthAuthProvider" />
</security:authentication-manager>
```

We've assumed here that the security namespace is being used for configuration. It's also assumed that you have added a `UserDetailsService` (called "userDetailsService") to your configuration to load the user's roles.

Java EE Container Authentication

The class `J2eePreAuthenticatedProcessingFilter` will extract the username from the `userPrincipal` property of the `HttpServletRequest`. Use of this filter would usually be combined with the use of Java EE roles as described above in J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource.

There is a sample application in the samples project which uses this approach, so get hold of the code from GitHub and have a look at the application context file if you are interested.

10.16. Java Authentication and Authorization Service (JAAS) Provider

10.16.1. Overview

Spring Security provides a package able to delegate authentication requests to the Java Authentication and Authorization Service (JAAS). This package is discussed in detail below.

10.16.2. AbstractJaasAuthenticationProvider

The `AbstractJaasAuthenticationProvider` is the basis for the provided JAAS `AuthenticationProvider` implementations. Subclasses must implement a method that creates the `LoginContext`. The `AbstractJaasAuthenticationProvider` has a number of dependencies that can be injected into it that are discussed below.

JAAS CallbackHandler

Most JAAS `LoginModule` s require a callback of some sort. These callbacks are usually used to obtain the username and password from the user.

In a Spring Security deployment, Spring Security is responsible for this user interaction (via the authentication mechanism). Thus, by the time the authentication request is delegated through to JAAS, Spring Security's authentication mechanism will already have fully-populated an `Authentication` object containing all the information required by the JAAS `LoginModule`.

Therefore, the JAAS package for Spring Security provides two default callback handlers, `JaasNameCallbackHandler` and `JaasPasswordCallbackHandler`. Each of these callback handlers implement `JaasAuthenticationCallbackHandler`. In most cases these callback handlers can simply be used without understanding the internal mechanics.

For those needing full control over the callback behavior, internally `AbstractJaasAuthenticationProvider` wraps these `JaasAuthenticationCallbackHandler` s with an `InternalCallbackHandler`. The `InternalCallbackHandler` is the class that actually implements JAAS normal `CallbackHandler` interface. Any time that the JAAS `LoginModule` is used, it is passed a list of application context configured `InternalCallbackHandler` s. If the `LoginModule` requests a callback against the `InternalCallbackHandler` s, the callback is in-turn passed to the `JaasAuthenticationCallbackHandler` s being wrapped.

JAAS AuthorityGranter

JAAS works with principals. Even "roles" are represented as principals in JAAS. Spring Security, on the other hand, works with `Authentication` objects. Each `Authentication` object contains a single principal, and multiple `GrantedAuthority` s. To facilitate mapping between these different concepts, Spring Security's JAAS package includes an `AuthorityGranter` interface.

An `AuthorityGranter` is responsible for inspecting a JAAS principal and returning a set of `String` s, representing the authorities assigned to the principal. For each returned authority string, the `AbstractJaasAuthenticationProvider` creates a `JaasGrantedAuthority` (which implements Spring Security's `GrantedAuthority` interface) containing the authority string and the JAAS principal that the `AuthorityGranter` was passed. The `AbstractJaasAuthenticationProvider` obtains the JAAS principals by firstly successfully authenticating the user's credentials using the JAAS `LoginModule`, and then accessing the `LoginContext` it returns. A call to `LoginContext.getSubject().getPrincipals()` is made, with each resulting principal passed to each `AuthorityGranter` defined against the

`AbstractJaasAuthenticationProvider.setAuthorityGranters(List)` property.

Spring Security does not include any production `AuthorityGranter` s given that every JAAS principal has an implementation-specific meaning. However, there is a `TestAuthorityGranter` in the unit tests that demonstrates a simple `AuthorityGranter` implementation.

10.16.3. DefaultJaasAuthenticationProvider

The `DefaultJaasAuthenticationProvider` allows a JAAS `Configuration` object to be injected into it as a dependency. It then creates a `LoginContext` using the injected JAAS `Configuration`. This means that `DefaultJaasAuthenticationProvider` is not bound any particular implementation of `Configuration` as `JaasAuthenticationProvider` is.

InMemoryConfiguration

In order to make it easy to inject a `Configuration` into `DefaultJaasAuthenticationProvider`, a default in-memory implementation named `InMemoryConfiguration` is provided. The implementation constructor accepts a `Map` where each key represents a login configuration name and the value represents an `Array` of `AppConfigurationEntry` s. `InMemoryConfiguration` also supports a default `Array` of `AppConfigurationEntry` objects that will be used if no mapping is found within the provided `Map`. For details, refer to the class level javadoc of `InMemoryConfiguration`.

DefaultJaasAuthenticationProvider Example Configuration

While the Spring configuration for `InMemoryConfiguration` can be more verbose than the standard JAAS configuration files, using it in conjunction with `DefaultJaasAuthenticationProvider` is more flexible than `JaasAuthenticationProvider` since it not dependant on the default `Configuration` implementation.

An example configuration of `DefaultJaasAuthenticationProvider` using `InMemoryConfiguration` is provided below. Note that custom implementations of `Configuration` can easily be injected into `DefaultJaasAuthenticationProvider` as well.

```
<bean id="jaasAuthProvider"
class="org.springframework.security.authentication.jaas.DefaultJaasAuthenticationProvi
der">
<property name="configuration">
<bean
class="org.springframework.security.authentication.jaas.memory.InMemoryConfiguration">
<constructor-arg>
    <map>
    <!--
    SPRINGSECURITY is the default loginContextName
    for AbstractJaasAuthenticationProvider
    -->
    <entry key="SPRINGSECURITY">
    <array>
    <bean class="javax.security.auth.login.AppConfigurationEntry">
        <constructor-arg value="sample.SampleLoginModule" />
        <constructor-arg>
        <util:constant static-field=

"javax.security.auth.login.AppConfigurationEntry$LoginModuleControlFlag.REQUIRED"/>
        </constructor-arg>
        <constructor-arg>
        <map></map>
        </constructor-arg>
        </bean>
    </array>
    </entry>
    </map>
    </constructor-arg>
</bean>
</property>
<property name="authorityGranters">
<list>
    <!-- You will need to write your own implementation of AuthorityGranter -->
    <bean
class="org.springframework.security.authentication.jaas.TestAuthorityGranter"/>
</list>
</property>
</bean>
```

10.16.4. JaasAuthenticationProvider

The `JaasAuthenticationProvider` assumes the default `Configuration` is an instance of ConfigFile. This assumption is made in order to attempt to update the `Configuration`. The `JaasAuthenticationProvider` then uses the default `Configuration` to create the `LoginContext`.

Let's assume we have a JAAS login configuration file, `/WEB-INF/login.conf`, with the following contents:

```
JAASTest {
    sample.SampleLoginModule required;
};
```

Like all Spring Security beans, the `JaasAuthenticationProvider` is configured via the application context. The following definitions would correspond to the above JAAS login configuration file:

```
<bean id="jaasAuthenticationProvider"
class="org.springframework.security.authentication.jaas.JaasAuthenticationProvider">
<property name="loginConfig" value="/WEB-INF/login.conf"/>
<property name="loginContextName" value="JAASTest"/>
<property name="callbackHandlers">
<list>
<bean
    class="org.springframework.security.authentication.jaas.JaasNameCallbackHandler"/>
<bean

class="org.springframework.security.authentication.jaas.JaasPasswordCallbackHandler"/>
</list>
</property>
<property name="authorityGranters">
    <list>
    <bean
class="org.springframework.security.authentication.jaas.TestAuthorityGranter"/>
    </list>
</property>
</bean>
```

10.16.5. Running as a Subject

If configured, the `JaasApiIntegrationFilter` will attempt to run as the `Subject` on the `JaasAuthenticationToken`. This means that the `Subject` can be accessed using:

```
Subject subject = Subject.getSubject(AccessController.getContext());
```

This integration can easily be configured using the jaas-api-provision attribute. This feature is useful when integrating with legacy or external API's that rely on the JAAS Subject being populated.

10.17. CAS Authentication

10.17.1. Overview

JA-SIG produces an enterprise-wide single sign on system known as CAS. Unlike other initiatives, JA-SIG's Central Authentication Service is open source, widely used, simple to understand, platform independent, and supports proxy capabilities. Spring Security fully supports CAS, and provides an easy migration path from single-application deployments of Spring Security through to multiple-

application deployments secured by an enterprise-wide CAS server.

You can learn more about CAS at https://www.apereo.org. You will also need to visit this site to download the CAS Server files.

10.17.2. How CAS Works

Whilst the CAS web site contains documents that detail the architecture of CAS, we present the general overview again here within the context of Spring Security. Spring Security 3.x supports CAS 3. At the time of writing, the CAS server was at version 3.4.

Somewhere in your enterprise you will need to setup a CAS server. The CAS server is simply a standard WAR file, so there isn't anything difficult about setting up your server. Inside the WAR file you will customise the login and other single sign on pages displayed to users.

When deploying a CAS 3.4 server, you will also need to specify an `AuthenticationHandler` in the `deployerConfigContext.xml` included with CAS. The `AuthenticationHandler` has a simple method that returns a boolean as to whether a given set of Credentials is valid. Your `AuthenticationHandler` implementation will need to link into some type of backend authentication repository, such as an LDAP server or database. CAS itself includes numerous `AuthenticationHandler` s out of the box to assist with this. When you download and deploy the server war file, it is set up to successfully authenticate users who enter a password matching their username, which is useful for testing.

Apart from the CAS server itself, the other key players are of course the secure web applications deployed throughout your enterprise. These web applications are known as "services". There are three types of services. Those that authenticate service tickets, those that can obtain proxy tickets, and those that authenticate proxy tickets. Authenticating a proxy ticket differs because the list of proxies must be validated and often times a proxy ticket can be reused.

Spring Security and CAS Interaction Sequence

The basic interaction between a web browser, CAS server and a Spring Security-secured service is as follows:

- The web user is browsing the service's public pages. CAS or Spring Security is not involved.
- The user eventually requests a page that is either secure or one of the beans it uses is secure. Spring Security's `ExceptionTranslationFilter` will detect the `AccessDeniedException` or `AuthenticationException`.
- Because the user's `Authentication` object (or lack thereof) caused an `AuthenticationException`, the `ExceptionTranslationFilter` will call the configured `AuthenticationEntryPoint`. If using CAS, this will be the `CasAuthenticationEntryPoint` class.
- The `CasAuthenticationEntryPoint` will redirect the user's browser to the CAS server. It will also indicate a `service` parameter, which is the callback URL for the Spring Security service (your application). For example, the URL to which the browser is redirected might be https://my.company.com/cas/login?service=https%3A%2F%2Fserver3.company.com%2Fwebapp%2Flogin/cas.
- After the user's browser redirects to CAS, they will be prompted for their username and password. If the user presents a session cookie which indicates they've previously logged on,

they will not be prompted to login again (there is an exception to this procedure, which we'll cover later). CAS will use the `PasswordHandler` (or `AuthenticationHandler` if using CAS 3.0) discussed above to decide whether the username and password is valid.

- Upon successful login, CAS will redirect the user's browser back to the original service. It will also include a `ticket` parameter, which is an opaque string representing the "service ticket". Continuing our earlier example, the URL the browser is redirected to might be https://server3.company.com/webapp/login/cas?ticket=ST-0-ER94xMJmn6pha35CQRoZ.
- Back in the service web application, the `CasAuthenticationFilter` is always listening for requests to `/login/cas` (this is configurable, but we'll use the defaults in this introduction). The processing filter will construct a `UsernamePasswordAuthenticationToken` representing the service ticket. The principal will be equal to `CasAuthenticationFilter.CAS_STATEFUL_IDENTIFIER`, whilst the credentials will be the service ticket opaque value. This authentication request will then be handed to the configured `AuthenticationManager`.
- The `AuthenticationManager` implementation will be the `ProviderManager`, which is in turn configured with the `CasAuthenticationProvider`. The `CasAuthenticationProvider` only responds to `UsernamePasswordAuthenticationToken` s containing the CAS-specific principal (such as `CasAuthenticationFilter.CAS_STATEFUL_IDENTIFIER`) and `CasAuthenticationToken` s (discussed later).
- `CasAuthenticationProvider` will validate the service ticket using a `TicketValidator` implementation. This will typically be a `Cas20ServiceTicketValidator` which is one of the classes included in the CAS client library. In the event the application needs to validate proxy tickets, the `Cas20ProxyTicketValidator` is used. The `TicketValidator` makes an HTTPS request to the CAS server in order to validate the service ticket. It may also include a proxy callback URL, which is included in this example: https://my.company.com/cas/proxyValidate?service=https%3A%2F%2Fserver3.company.com%2Fwebapp%2Flogin/cas&ticket=ST-0-ER94xMJmn6pha35CQRoZ&pgtUrl=https://server3.company.com/webapp/login/cas/proxyreceptor.
- Back on the CAS server, the validation request will be received. If the presented service ticket matches the service URL the ticket was issued to, CAS will provide an affirmative response in XML indicating the username. If any proxy was involved in the authentication (discussed below), the list of proxies is also included in the XML response.
- [OPTIONAL] If the request to the CAS validation service included the proxy callback URL (in the `pgtUrl` parameter), CAS will include a `pgtIou` string in the XML response. This `pgtIou` represents a proxy-granting ticket IOU. The CAS server will then create its own HTTPS connection back to the `pgtUrl`. This is to mutually authenticate the CAS server and the claimed service URL. The HTTPS connection will be used to send a proxy granting ticket to the original web application. For example, https://server3.company.com/webapp/login/cas/proxyreceptor?pgtIou=PGTIOU-0-R0zlgrl4pdAQwBvJWO3vnNpevwqStbSGcq3vKB2SqSFFRnjPHt&pgtId=PGT-1-si9YkkHLrtACBo64rmsi3v2nf7cpCResXg5MpESZFArbaZiOKH.
- The `Cas20TicketValidator` will parse the XML received from the CAS server. It will return to the `CasAuthenticationProvider` a `TicketResponse`, which includes the username (mandatory), proxy list (if any were involved), and proxy-granting ticket IOU (if the proxy callback was requested).
- Next `CasAuthenticationProvider` will call a configured `CasProxyDecider`. The `CasProxyDecider` indicates whether the proxy list in the `TicketResponse` is acceptable to the service. Several implementations are provided with Spring Security: `RejectProxyTickets`, `AcceptAnyCasProxy` and

`NamedCasProxyDecider`. These names are largely self-explanatory, except `NamedCasProxyDecider` which allows a `List` of trusted proxies to be provided.

- `CasAuthenticationProvider` will next request a `AuthenticationUserDetailsService` to load the `GrantedAuthority` objects that apply to the user contained in the `Assertion`.
- If there were no problems, `CasAuthenticationProvider` constructs a `CasAuthenticationToken` including the details contained in the `TicketResponse` and the `GrantedAuthority`s.
- Control then returns to `CasAuthenticationFilter`, which places the created `CasAuthenticationToken` in the security context.
- The user's browser is redirected to the original page that caused the `AuthenticationException` (or a custom destination depending on the configuration).

It's good that you're still here! Let's now look at how this is configured

10.17.3. Configuration of CAS Client

The web application side of CAS is made easy due to Spring Security. It is assumed you already know the basics of using Spring Security, so these are not covered again below. We'll assume a namespace based configuration is being used and add in the CAS beans as required. Each section builds upon the previous section. A full CAS sample application can be found in the Spring Security Samples.

Service Ticket Authentication

This section describes how to setup Spring Security to authenticate Service Tickets. Often times this is all a web application requires. You will need to add a `ServiceProperties` bean to your application context. This represents your CAS service:

```
<bean id="serviceProperties"
    class="org.springframework.security.cas.ServiceProperties">
<property name="service"
    value="https://localhost:8443/cas-sample/login/cas"/>
<property name="sendRenew" value="false"/>
</bean>
```

The `service` must equal a URL that will be monitored by the `CasAuthenticationFilter`. The `sendRenew` defaults to false, but should be set to true if your application is particularly sensitive. What this parameter does is tell the CAS login service that a single sign on login is unacceptable. Instead, the user will need to re-enter their username and password in order to gain access to the service.

The following beans should be configured to commence the CAS authentication process (assuming you're using a namespace configuration):

```
<security:http entry-point-ref="casEntryPoint">
...
<security:custom-filter position="CAS_FILTER" ref="casFilter" />
</security:http>

<bean id="casFilter"
    class="org.springframework.security.cas.web.CasAuthenticationFilter">
<property name="authenticationManager" ref="authenticationManager"/>
</bean>

<bean id="casEntryPoint"
    class="org.springframework.security.cas.web.CasAuthenticationEntryPoint">
<property name="loginUrl" value="https://localhost:9443/cas/login"/>
<property name="serviceProperties" ref="serviceProperties"/>
</bean>
```

For CAS to operate, the `ExceptionTranslationFilter` must have its `authenticationEntryPoint` property set to the `CasAuthenticationEntryPoint` bean. This can easily be done using entry-point-ref as is done in the example above. The `CasAuthenticationEntryPoint` must refer to the `ServiceProperties` bean (discussed above), which provides the URL to the enterprise's CAS login server. This is where the user's browser will be redirected.

The `CasAuthenticationFilter` has very similar properties to the `UsernamePasswordAuthenticationFilter` (used for form-based logins). You can use these properties to customize things like behavior for authentication success and failure.

Next you need to add a `CasAuthenticationProvider` and its collaborators:

```
<security:authentication-manager alias="authenticationManager">
<security:authentication-provider ref="casAuthenticationProvider" />
</security:authentication-manager>

<bean id="casAuthenticationProvider"
    class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
<property name="authenticationUserDetailsService">
    <bean
class="org.springframework.security.core.userdetails.UserDetailsByNameServiceWrapper">
    <constructor-arg ref="userService" />
    </bean>
</property>
<property name="serviceProperties" ref="serviceProperties" />
<property name="ticketValidator">
    <bean class="org.jasig.cas.client.validation.Cas20ServiceTicketValidator">
    <constructor-arg index="0" value="https://localhost:9443/cas" />
    </bean>
</property>
<property name="key" value="an_id_for_this_auth_provider_only"/>
</bean>

<security:user-service id="userService">
<!-- Password is prefixed with {noop} to indicate to DelegatingPasswordEncoder that
NoOpPasswordEncoder should be used.
This is not safe for production, but makes reading
in samples easier.
Normally passwords should be hashed using BCrypt -->
<security:user name="joe" password="{noop}joe" authorities="ROLE_USER" />
...
</security:user-service>
```

The `CasAuthenticationProvider` uses a `UserDetailsService` instance to load the authorities for a user, once they have been authenticated by CAS. We've shown a simple in-memory setup here. Note that the `CasAuthenticationProvider` does not actually use the password for authentication, but it does use the authorities.

The beans are all reasonably self-explanatory if you refer back to the How CAS Works section.

This completes the most basic configuration for CAS. If you haven't made any mistakes, your web application should happily work within the framework of CAS single sign on. No other parts of Spring Security need to be concerned about the fact CAS handled authentication. In the following sections we will discuss some (optional) more advanced configurations.

Single Logout

The CAS protocol supports Single Logout and can be easily added to your Spring Security configuration. Below are updates to the Spring Security configuration that handle Single Logout

```
<security:http entry-point-ref="casEntryPoint">
...
<security:logout logout-success-url="/cas-logout.jsp"/>
<security:custom-filter ref="requestSingleLogoutFilter" before="LOGOUT_FILTER"/>
<security:custom-filter ref="singleLogoutFilter" before="CAS_FILTER"/>
</security:http>

<!-- This filter handles a Single Logout Request from the CAS Server -->
<bean id="singleLogoutFilter"
class="org.jasig.cas.client.session.SingleSignOutFilter"/>

<!-- This filter redirects to the CAS Server to signal Single Logout should be
performed -->
<bean id="requestSingleLogoutFilter"
    class="org.springframework.security.web.authentication.logout.LogoutFilter">
<constructor-arg value="https://localhost:9443/cas/logout"/>
<constructor-arg>
    <bean class=

"org.springframework.security.web.authentication.logout.SecurityContextLogoutHandler"/
>
</constructor-arg>
<property name="filterProcessesUrl" value="/logout/cas"/>
</bean>
```

The `logout` element logs the user out of the local application, but does not end the session with the CAS server or any other applications that have been logged into. The `requestSingleLogoutFilter` filter will allow the URL of `/spring_security_cas_logout` to be requested to redirect the application to the configured CAS Server logout URL. Then the CAS Server will send a Single Logout request to all the services that were signed into. The `singleLogoutFilter` handles the Single Logout request by looking up the `HttpSession` in a static `Map` and then invalidating it.

It might be confusing why both the `logout` element and the `singleLogoutFilter` are needed. It is considered best practice to logout locally first since the `SingleSignOutFilter` just stores the `HttpSession` in a static `Map` in order to call invalidate on it. With the configuration above, the flow of logout would be:

- The user requests `/logout` which would log the user out of the local application and send the user to the logout success page.
- The logout success page, `/cas-logout.jsp`, should instruct the user to click a link pointing to `/logout/cas` in order to logout out of all applications.
- When the user clicks the link, the user is redirected to the CAS single logout URL (https://localhost:9443/cas/logout).
- On the CAS Server side, the CAS single logout URL then submits single logout requests to all the CAS Services. On the CAS Service side, JASIG's `SingleSignOutFilter` processes the logout request by invalidating the original session.

The next step is to add the following to your web.xml

```
<filter>
<filter-name>characterEncodingFilter</filter-name>
<filter-class>
    org.springframework.web.filter.CharacterEncodingFilter
</filter-class>
<init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
</init-param>
</filter>
<filter-mapping>
<filter-name>characterEncodingFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
<listener>
<listener-class>
    org.jasig.cas.client.session.SingleSignOutHttpSessionListener
</listener-class>
</listener>
```

When using the SingleSignOutFilter you might encounter some encoding issues. Therefore it is recommended to add the `CharacterEncodingFilter` to ensure that the character encoding is correct when using the `SingleSignOutFilter`. Again, refer to JASIG's documentation for details. The `SingleSignOutHttpSessionListener` ensures that when an `HttpSession` expires, the mapping used for single logout is removed.

Authenticating to a Stateless Service with CAS

This section describes how to authenticate to a service using CAS. In other words, this section discusses how to setup a client that uses a service that authenticates with CAS. The next section describes how to setup a stateless service to Authenticate using CAS.

Configuring CAS to Obtain Proxy Granting Tickets

In order to authenticate to a stateless service, the application needs to obtain a proxy granting ticket (PGT). This section describes how to configure Spring Security to obtain a PGT building upon thencas-st[Service Ticket Authentication] configuration.

The first step is to include a `ProxyGrantingTicketStorage` in your Spring Security configuration. This is used to store PGT's that are obtained by the `CasAuthenticationFilter` so that they can be used to obtain proxy tickets. An example configuration is shown below

```
<!--
NOTE: In a real application you should not use an in memory implementation.
You will also want to ensure to clean up expired tickets by calling
ProxyGrantingTicketStorage.cleanup()
-->
<bean id="pgtStorage"
class="org.jasig.cas.client.proxy.ProxyGrantingTicketStorageImpl"/>
```

The next step is to update the `CasAuthenticationProvider` to be able to obtain proxy tickets. To do this replace the `Cas20ServiceTicketValidator` with a `Cas20ProxyTicketValidator`. The `proxyCallbackUrl` should be set to a URL that the application will receive PGT's at. Last, the configuration should also reference the `ProxyGrantingTicketStorage` so it can use a PGT to obtain proxy tickets. You can find an example of the configuration changes that should be made below.

```
<bean id="casAuthenticationProvider"
    class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
...
<property name="ticketValidator">
    <bean class="org.jasig.cas.client.validation.Cas20ProxyTicketValidator">
    <constructor-arg value="https://localhost:9443/cas"/>
        <property name="proxyCallbackUrl"
        value="https://localhost:8443/cas-sample/login/cas/proxyreceptor"/>
    <property name="proxyGrantingTicketStorage" ref="pgtStorage"/>
    </bean>
</property>
</bean>
```

The last step is to update the `CasAuthenticationFilter` to accept PGT and to store them in the `ProxyGrantingTicketStorage`. It is important the `proxyReceptorUrl` matches the `proxyCallbackUrl` of the `Cas20ProxyTicketValidator`. An example configuration is shown below.

```
<bean id="casFilter"
        class="org.springframework.security.cas.web.CasAuthenticationFilter">
    ...
    <property name="proxyGrantingTicketStorage" ref="pgtStorage"/>
    <property name="proxyReceptorUrl" value="/login/cas/proxyreceptor"/>
</bean>
```

Calling a Stateless Service Using a Proxy Ticket

Now that Spring Security obtains PGTs, you can use them to create proxy tickets which can be used to authenticate to a stateless service. The CAS sample application contains a working example in the `ProxyTicketSampleServlet`. Example code can be found below:

Java

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
// NOTE: The CasAuthenticationToken can also be obtained using
// SecurityContextHolder.getContext().getAuthentication()
final CasAuthenticationToken token = (CasAuthenticationToken)
request.getUserPrincipal();
// proxyTicket could be reused to make calls to the CAS service even if the
// target url differs
final String proxyTicket =
token.getAssertion().getPrincipal().getProxyTicketFor(targetUrl);

// Make a remote call using the proxy ticket
final String serviceUrl = targetUrl+"?ticket="+URLEncoder.encode(proxyTicket,
"UTF-8");
String proxyResponse = CommonUtils.getResponseFromServer(serviceUrl, "UTF-8");
...
}
```

Kotlin

```
protected fun doGet(request: HttpServletRequest, response: HttpServletResponse?) {
    // NOTE: The CasAuthenticationToken can also be obtained using
    // SecurityContextHolder.getContext().getAuthentication()
    val token = request.userPrincipal as CasAuthenticationToken
    // proxyTicket could be reused to make calls to the CAS service even if the
    // target url differs
    val proxyTicket = token.assertion.principal.getProxyTicketFor(targetUrl)

    // Make a remote call using the proxy ticket
    val serviceUrl: String = targetUrl + "?ticket=" +
URLEncoder.encode(proxyTicket, "UTF-8")
    val proxyResponse = CommonUtils.getResponseFromServer(serviceUrl, "UTF-8")
}
```

Proxy Ticket Authentication

The `CasAuthenticationProvider` distinguishes between stateful and stateless clients. A stateful client is considered any that submits to the `filterProcessUrl` of the `CasAuthenticationFilter`. A stateless client is any that presents an authentication request to `CasAuthenticationFilter` on a URL other than the `filterProcessUrl`.

Because remoting protocols have no way of presenting themselves within the context of an `HttpSession`, it isn't possible to rely on the default practice of storing the security context in the session between requests. Furthermore, because the CAS server invalidates a ticket after it has been validated by the `TicketValidator`, presenting the same proxy ticket on subsequent requests will not work.

One obvious option is to not use CAS at all for remoting protocol clients. However, this would eliminate many of the desirable features of CAS. As a middle-ground, the `CasAuthenticationProvider` uses a `StatelessTicketCache`. This is used solely for stateless clients which use a principal equal to `CasAuthenticationFilter.CAS_STATELESS_IDENTIFIER`. What happens is the `CasAuthenticationProvider` will store the resulting `CasAuthenticationToken` in the `StatelessTicketCache`, keyed on the proxy ticket. Accordingly, remoting protocol clients can present the same proxy ticket and the `CasAuthenticationProvider` will not need to contact the CAS server for validation (aside from the first request). Once authenticated, the proxy ticket could be used for URLs other than the original target service.

This section builds upon the previous sections to accommodate proxy ticket authentication. The first step is to specify to authenticate all artifacts as shown below.

```
<bean id="serviceProperties"
    class="org.springframework.security.cas.ServiceProperties">
...
<property name="authenticateAllArtifacts" value="true"/>
</bean>
```

The next step is to specify `serviceProperties` and the `authenticationDetailsSource` for the `CasAuthenticationFilter`. The `serviceProperties` property instructs the `CasAuthenticationFilter` to attempt to authenticate all artifacts instead of only ones present on the `filterProcessUrl`. The `ServiceAuthenticationDetailsSource` creates a `ServiceAuthenticationDetails` that ensures the current URL, based upon the `HttpServletRequest`, is used as the service URL when validating the ticket. The method for generating the service URL can be customized by injecting a custom `AuthenticationDetailsSource` that returns a custom `ServiceAuthenticationDetails`.

```
<bean id="casFilter"
    class="org.springframework.security.cas.web.CasAuthenticationFilter">
...
<property name="serviceProperties" ref="serviceProperties"/>
<property name="authenticationDetailsSource">
    <bean class=

"org.springframework.security.cas.web.authentication.ServiceAuthenticationDetailsSourc
e">
    <constructor-arg ref="serviceProperties"/>
    </bean>
</property>
</bean>
```

You will also need to update the `CasAuthenticationProvider` to handle proxy tickets. To do this replace the `Cas20ServiceTicketValidator` with a `Cas20ProxyTicketValidator`. You will need to configure the `statelessTicketCache` and which proxies you want to accept. You can find an example of the updates required to accept all proxies below.

```
<bean id="casAuthenticationProvider"
    class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
...
<property name="ticketValidator">
    <bean class="org.jasig.cas.client.validation.Cas20ProxyTicketValidator">
    <constructor-arg value="https://localhost:9443/cas"/>
    <property name="acceptAnyProxy" value="true"/>
    </bean>
</property>
<property name="statelessTicketCache">
    <bean
class="org.springframework.security.cas.authentication.EhCacheBasedTicketCache">
    <property name="cache">
        <bean class="net.sf.ehcache.Cache"
            init-method="initialise" destroy-method="dispose">
        <constructor-arg value="casTickets"/>
        <constructor-arg value="50"/>
        <constructor-arg value="true"/>
        <constructor-arg value="false"/>
        <constructor-arg value="3600"/>
        <constructor-arg value="900"/>
        </bean>
    </property>
    </bean>
</property>
</bean>
```

10.18. X.509 Authentication

10.18.1. Overview

The most common use of X.509 certificate authentication is in verifying the identity of a server when using SSL, most commonly when using HTTPS from a browser. The browser will automatically check that the certificate presented by a server has been issued (ie digitally signed) by one of a list of trusted certificate authorities which it maintains.

You can also use SSL with "mutual authentication"; the server will then request a valid certificate from the client as part of the SSL handshake. The server will authenticate the client by checking that its certificate is signed by an acceptable authority. If a valid certificate has been provided, it can be obtained through the servlet API in an application. Spring Security X.509 module extracts the certificate using a filter. It maps the certificate to an application user and loads that user's set of granted authorities for use with the standard Spring Security infrastructure.

You should be familiar with using certificates and setting up client authentication for your servlet container before attempting to use it with Spring Security. Most of the work is in creating and installing suitable certificates and keys. For example, if you're using Tomcat then read the instructions here https://tomcat.apache.org/tomcat-9.0-doc/ssl-howto.html. It's important that you get this working before trying it out with Spring Security

10.18.2. Adding X.509 Authentication to Your Web Application

Enabling X.509 client authentication is very straightforward. Just add the `<x509/>` element to your http security namespace configuration.

```
<http>
...
    <x509 subject-principal-regex="CN=(.*?)," user-service-ref="userService"/>;
</http>
```

The element has two optional attributes:

- `subject-principal-regex`. The regular expression used to extract a username from the certificate's subject name. The default value is shown above. This is the username which will be passed to the `UserDetailsService` to load the authorities for the user.
- `user-service-ref`. This is the bean Id of the `UserDetailsService` to be used with X.509. It isn't needed if there is only one defined in your application context.

The `subject-principal-regex` should contain a single group. For example the default expression "CN=(.*?)," matches the common name field. So if the subject name in the certificate is "CN=Jimi Hendrix, OU=...", this will give a user name of "Jimi Hendrix". The matches are case insensitive. So "emailAddress=(.*?)," will match "EMAILADDRESS=jimi@hendrix.org,CN=..." giving a user name "jimi@hendrix.org". If the client presents a certificate and a valid username is successfully extracted, then there should be a valid `Authentication` object in the security context. If no certificate is found, or no corresponding user could be found then the security context will remain empty. This means that you can easily use X.509 authentication with other options such as a form-based login.

10.18.3. Setting up SSL in Tomcat

There are some pre-generated certificates in the Spring Security Samples repository. You can use these to enable SSL for testing if you don't want to generate your own. The file `server.jks` contains the server certificate, private key and the issuing certificate authority certificate. There are also some client certificate files for the users from the sample applications. You can install these in your browser to enable SSL client authentication.

To run tomcat with SSL support, drop the `server.jks` file into the tomcat `conf` directory and add the following connector to the `server.xml` file

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true" scheme="https"
secure="true"
            clientAuth="true" sslProtocol="TLS"
            keystoreFile="${catalina.home}/conf/server.jks"
            keystoreType="JKS" keystorePass="password"
            truststoreFile="${catalina.home}/conf/server.jks"
            truststoreType="JKS" truststorePass="password"
/>
```

clientAuth can also be set to want if you still want SSL connections to succeed even if the client doesn't provide a certificate. Clients which don't present a certificate won't be able to access any objects secured by Spring Security unless you use a non-X.509 authentication mechanism, such as form authentication.

10.19. Run-As Authentication Replacement

10.19.1. Overview

The AbstractSecurityInterceptor is able to temporarily replace the Authentication object in the SecurityContext and SecurityContextHolder during the secure object callback phase. This only occurs if the original Authentication object was successfully processed by the AuthenticationManager and AccessDecisionManager. The RunAsManager will indicate the replacement Authentication object, if any, that should be used during the SecurityInterceptorCallback.

By temporarily replacing the Authentication object during the secure object callback phase, the secured invocation will be able to call other objects which require different authentication and authorization credentials. It will also be able to perform any internal security checks for specific GrantedAuthority objects. Because Spring Security provides a number of helper classes that automatically configure remoting protocols based on the contents of the SecurityContextHolder, these run-as replacements are particularly useful when calling remote web services

10.19.2. Configuration

A RunAsManager interface is provided by Spring Security:

```
Authentication buildRunAs(Authentication authentication, Object object,
    List<ConfigAttribute> config);

boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

The first method returns the Authentication object that should replace the existing Authentication object for the duration of the method invocation. If the method returns null, it indicates no replacement should be made. The second method is used by the AbstractSecurityInterceptor as part of its startup validation of configuration attributes. The supports(Class) method is called by a security interceptor implementation to ensure the configured RunAsManager supports the type of secure object that the security interceptor will present.

One concrete implementation of a RunAsManager is provided with Spring Security. The RunAsManagerImpl class returns a replacement RunAsUserToken if any ConfigAttribute starts with RUN_AS_. If any such ConfigAttribute is found, the replacement RunAsUserToken will contain the same principal, credentials and granted authorities as the original Authentication object, along with a new SimpleGrantedAuthority for each RUN_AS_ ConfigAttribute. Each new SimpleGrantedAuthority will be prefixed with ROLE_, followed by the RUN_AS ConfigAttribute. For example, a RUN_AS_SERVER will result in the replacement RunAsUserToken containing a ROLE_RUN_AS_SERVER granted authority.

The replacement `RunAsUserToken` is just like any other `Authentication` object. It needs to be authenticated by the `AuthenticationManager`, probably via delegation to a suitable `AuthenticationProvider`. The `RunAsImplAuthenticationProvider` performs such authentication. It simply accepts as valid any `RunAsUserToken` presented.

To ensure malicious code does not create a `RunAsUserToken` and present it for guaranteed acceptance by the `RunAsImplAuthenticationProvider`, the hash of a key is stored in all generated tokens. The `RunAsManagerImpl` and `RunAsImplAuthenticationProvider` is created in the bean context with the same key:

```
<bean id="runAsManager"
    class="org.springframework.security.access.intercept.RunAsManagerImpl">
<property name="key" value="my_run_as_password"/>
</bean>

<bean id="runAsAuthenticationProvider"

class="org.springframework.security.access.intercept.RunAsImplAuthenticationProvider">
<property name="key" value="my_run_as_password"/>
</bean>
```

By using the same key, each `RunAsUserToken` can be validated it was created by an approved `RunAsManagerImpl`. The `RunAsUserToken` is immutable after creation for security reasons

10.20. Handling Logouts

10.20.1. Logout Java/Kotlin Configuration

When using the `WebSecurityConfigurerAdapter`, logout capabilities are automatically applied. The default is that accessing the URL `/logout` will log the user out by:

- Invalidating the HTTP Session
- Cleaning up any RememberMe authentication that was configured
- Clearing the `SecurityContextHolder`
- Redirect to `/login?logout`

Similar to configuring login capabilities, however, you also have various options to further customize your logout requirements:

Example 87. Logout Configuration

Java

```
protected void configure(HttpSecurity http) throws Exception {
    http
        .logout(logout -> logout                                    ①
            .logoutUrl("/my/logout")                               ②
            .logoutSuccessUrl("/my/index")                         ③
            .logoutSuccessHandler(logoutSuccessHandler)            ④
            .invalidateHttpSession(true)                           ⑤
            .addLogoutHandler(logoutHandler)                       ⑥
            .deleteCookies(cookieNamesToClear)                     ⑦
        )
        ...
}
```

Kotlin

```
override fun configure(http: HttpSecurity) {
    http {
        logout {
            logoutUrl = "/my/logout"                               ①
            logoutSuccessUrl = "/my/index"                         ②
            logoutSuccessHandler = customLogoutSuccessHandler      ③
            invalidateHttpSession = true                           ④
            addLogoutHandler(logoutHandler)                        ⑤
            deleteCookies(cookieNamesToClear)                      ⑥
        }
    }
}
```

① Provides logout support. This is automatically applied when using `WebSecurityConfigurerAdapter`.

② The URL that triggers log out to occur (default is `/logout`). If CSRF protection is enabled (default), then the request must also be a POST. For more information, please consult the JavaDoc.

③ The URL to redirect to after logout has occurred. The default is `/login?logout`. For more information, please consult the JavaDoc.

④ Let's you specify a custom `LogoutSuccessHandler`. If this is specified, `logoutSuccessUrl()` is ignored. For more information, please consult the JavaDoc.

⑤ Specify whether to invalidate the `HttpSession` at the time of logout. This is **true** by default. Configures the `SecurityContextLogoutHandler` under the covers. For more information, please consult the JavaDoc.

⑥ Adds a `LogoutHandler`. `SecurityContextLogoutHandler` is added as the last `LogoutHandler` by default.

⑦ Allows specifying the names of cookies to be removed on logout success. This is a shortcut for adding a `CookieClearingLogoutHandler` explicitly.

Logouts can of course also be configured using the XML Namespace notation. Please see the documentation for the logout element in the Spring Security XML Namespace section for further details.

Generally, in order to customize logout functionality, you can add `LogoutHandler` and/or `LogoutSuccessHandler` implementations. For many common scenarios, these handlers are applied under the covers when using the fluent API.

10.20.2. Logout XML Configuration

The `logout` element adds support for logging out by navigating to a particular URL. The default logout URL is `/logout`, but you can set it to something else using the `logout-url` attribute. More information on other available attributes may be found in the namespace appendix.

10.20.3. LogoutHandler

Generally, `LogoutHandler` implementations indicate classes that are able to participate in logout handling. They are expected to be invoked to perform necessary clean-up. As such they should not throw exceptions. Various implementations are provided:

- PersistentTokenBasedRememberMeServices
- TokenBasedRememberMeServices
- CookieClearingLogoutHandler
- CsrfLogoutHandler
- SecurityContextLogoutHandler
- HeaderWriterLogoutHandler

Please see Remember-Me Interfaces and Implementations for details.

Instead of providing `LogoutHandler` implementations directly, the fluent API also provides shortcuts that provide the respective `LogoutHandler` implementations under the covers. E.g. `deleteCookies()` allows specifying the names of one or more cookies to be removed on logout success. This is a shortcut compared to adding a `CookieClearingLogoutHandler`.

10.20.4. LogoutSuccessHandler

The `LogoutSuccessHandler` is called after a successful logout by the `LogoutFilter`, to handle e.g. redirection or forwarding to the appropriate destination. Note that the interface is almost the same as the `LogoutHandler` but may raise an exception.

The following implementations are provided:

- SimpleUrlLogoutSuccessHandler
- HttpStatusReturningLogoutSuccessHandler

As mentioned above, you don't need to specify the `SimpleUrlLogoutSuccessHandler` directly. Instead, the fluent API provides a shortcut by setting the `logoutSuccessUrl()`. This will setup the

`SimpleUrlLogoutSuccessHandler` under the covers. The provided URL will be redirected to after a logout has occurred. The default is `/login?logout`.

The `HttpStatusReturningLogoutSuccessHandler` can be interesting in REST API type scenarios. Instead of redirecting to a URL upon the successful logout, this `LogoutSuccessHandler` allows you to provide a plain HTTP status code to be returned. If not configured a status code 200 will be returned by default.

10.20.5. Further Logout-Related References

- Logout Handling
- Testing Logout
- HttpServletRequest.logout()
- Remember-Me Interfaces and Implementations
- Logging Out in section CSRF Caveats
- Section Single Logout (CAS protocol)
- Documentation for the logout element in the Spring Security XML Namespace section

10.21. Authentication Events

For each authentication that succeeds or fails, a `AuthenticationSuccessEvent` or `AbstractAuthenticationFailureEvent` is fired, respectively.

To listen for these events, you must first publish an `AuthenticationEventPublisher`. Spring Security's `DefaultAuthenticationEventPublisher` will probably do fine:

Java

```
@Bean
public AuthenticationEventPublisher authenticationEventPublisher
        (ApplicationEventPublisher applicationEventPublisher) {
    return new DefaultAuthenticationEventPublisher(applicationEventPublisher);
}
```

Kotlin

```
@Bean
fun authenticationEventPublisher
        (applicationEventPublisher: ApplicationEventPublisher?):
AuthenticationEventPublisher {
    return DefaultAuthenticationEventPublisher(applicationEventPublisher)
}
```

Then, you can use Spring's `@EventListener` support:

Java

```
@Component
public class AuthenticationEvents {
    @EventListener
    public void onSuccess(AuthenticationSuccessEvent success) {
        // ...
    }

    @EventListener
    public void onFailure(AbstractAuthenticationFailureEvent failures) {
        // ...
    }
}
```

Kotlin

```
@Component
class AuthenticationEvents {
    @EventListener
    fun onSuccess(success: AuthenticationSuccessEvent?) {
        // ...
    }

    @EventListener
    fun onFailure(failures: AbstractAuthenticationFailureEvent?) {
        // ...
    }
}
```

While similar to `AuthenticationSuccessHandler` and `AuthenticationFailureHandler`, these are nice in that they can be used independently from the servlet API.

10.21.1. Adding Exception Mappings

`DefaultAuthenticationEventPublisher` by default will publish an `AbstractAuthenticationFailureEvent` for the following events:

Exception	Event
`BadCredentialsException`	`AuthenticationFailureBadCredentialsEvent`
`UsernameNotFoundException`	`AuthenticationFailureBadCredentialsEvent`
`AccountExpiredException`	`AuthenticationFailureExpiredEvent`
`ProviderNotFoundException`	`AuthenticationFailureProviderNotFoundEvent`
`DisabledException`	`AuthenticationFailureDisabledEvent`
`LockedException`	`AuthenticationFailureLockedEvent`

`AuthenticationServiceException`	`AuthenticationFailureServiceExceptionEvent`
`CredentialsExpiredException`	`AuthenticationFailureCredentialsExpiredEvent`
`InvalidBearerTokenException`	`AuthenticationFailureBadCredentialsEvent`

The publisher does an exact `Exception` match, which means that sub-classes of these exceptions won't also produce events.

To that end, you may want to supply additional mappings to the publisher via the `setAdditionalExceptionMappings` method:

Java

```
@Bean
public AuthenticationEventPublisher authenticationEventPublisher
        (ApplicationEventPublisher applicationEventPublisher) {
    Map<Class<? extends AuthenticationException>,
        Class<? extends AbstractAuthenticationFailureEvent>> mapping =
            Collections.singletonMap(FooException.class, FooEvent.class);
    AuthenticationEventPublisher authenticationEventPublisher =
        new DefaultAuthenticationEventPublisher(applicationEventPublisher);
    authenticationEventPublisher.setAdditionalExceptionMappings(mapping);
    return authenticationEventPublisher;
}
```

Kotlin

```
@Bean
fun authenticationEventPublisher
        (applicationEventPublisher: ApplicationEventPublisher?):
AuthenticationEventPublisher {
    val mapping: Map<Class<out AuthenticationException>, Class<out
AbstractAuthenticationFailureEvent>> =
            mapOf(Pair(FooException::class.java, FooEvent::class.java))
    val authenticationEventPublisher =
DefaultAuthenticationEventPublisher(applicationEventPublisher)
    authenticationEventPublisher.setAdditionalExceptionMappings(mapping)
    return authenticationEventPublisher
}
```

10.21.2. Default Event

And, you can supply a catch-all event to fire in the case of any `AuthenticationException`:

Java

```
@Bean
public AuthenticationEventPublisher authenticationEventPublisher
        (ApplicationEventPublisher applicationEventPublisher) {
    AuthenticationEventPublisher authenticationEventPublisher =
        new DefaultAuthenticationEventPublisher(applicationEventPublisher);
    authenticationEventPublisher.setDefaultAuthenticationFailureEvent
        (GenericAuthenticationFailureEvent.class);
    return authenticationEventPublisher;
}
```

Kotlin

```
@Bean
fun authenticationEventPublisher
        (applicationEventPublisher: ApplicationEventPublisher?):
AuthenticationEventPublisher {
    val authenticationEventPublisher =
DefaultAuthenticationEventPublisher(applicationEventPublisher)

authenticationEventPublisher.setDefaultAuthenticationFailureEvent(GenericAuthentic
ationFailureEvent::class.java)
    return authenticationEventPublisher
}
```

[2] It is also possible to obtain the server's IP address using a DNS lookup. This is not currently supported, but hopefully will be in a future version.

[3] Authentication by mechanisms which perform a redirect after authenticating (such as form-login) will not be detected by `SessionManagementFilter`, as the filter will not be invoked during the authenticating request. Session-management functionality has to be handled separately in these cases.

[4] Essentially, the username is not included in the cookie, to prevent exposing a valid login name unecessarily. There is a discussion on this in the comments section of this article.

[5] The use of the `key` property should not be regarded as providing any real security here. It is merely a book-keeping exercise. If you are sharing a `ProviderManager` which contains an `AnonymousAuthenticationProvider` in a scenario where it is possible for an authenticating client to construct the `Authentication` object (such as with RMI invocations), then a malicious client could submit an `AnonymousAuthenticationToken` which it had created itself (with chosen username and authority list). If the `key` is guessable or can be found out, then the token would be accepted by the anonymous provider. This isn't a problem with normal usage but if you are using RMI you would be best to use a customized `ProviderManager` which omits the anonymous provider rather than sharing the one you use for your HTTP authentication mechanisms.

Chapter 11. Authorization

The advanced authorization capabilities within Spring Security represent one of the most compelling reasons for its popularity. Irrespective of how you choose to authenticate - whether using a Spring Security-provided mechanism and provider, or integrating with a container or other non-Spring Security authentication authority - you will find the authorization services can be used within your application in a consistent and simple way.

In this part we'll explore the different `AbstractSecurityInterceptor` implementations, which were introduced in Part I. We then move on to explore how to fine-tune authorization through use of domain access control lists.

11.1. Authorization Architecture

11.1.1. Authorities

`Authentication`, discusses how all `Authentication` implementations store a list of `GrantedAuthority` objects. These represent the authorities that have been granted to the principal. the `GrantedAuthority` objects are inserted into the `Authentication` object by the `AuthenticationManager` and are later read by `AccessDecisionManager` s when making authorization decisions.

`GrantedAuthority` is an interface with only one method:

```
String getAuthority();
```

This method allows `AccessDecisionManager` s to obtain a precise `String` representation of the `GrantedAuthority`. By returning a representation as a `String`, a `GrantedAuthority` can be easily "read" by most `AccessDecisionManager` s. If a `GrantedAuthority` cannot be precisely represented as a `String`, the `GrantedAuthority` is considered "complex" and `getAuthority()` must return `null`.

An example of a "complex" `GrantedAuthority` would be an implementation that stores a list of operations and authority thresholds that apply to different customer account numbers. Representing this complex `GrantedAuthority` as a `String` would be quite difficult, and as a result the `getAuthority()` method should return `null`. This will indicate to any `AccessDecisionManager` that it will need to specifically support the `GrantedAuthority` implementation in order to understand its contents.

Spring Security includes one concrete `GrantedAuthority` implementation, `SimpleGrantedAuthority`. This allows any user-specified `String` to be converted into a `GrantedAuthority`. All `AuthenticationProvider` s included with the security architecture use `SimpleGrantedAuthority` to populate the `Authentication` object.

11.1.2. Pre-Invocation Handling

Spring Security provides interceptors which control access to secure objects such as method invocations or web requests. A pre-invocation decision on whether the invocation is allowed to proceed is made by the `AccessDecisionManager`.

The AccessDecisionManager

The `AccessDecisionManager` is called by the `AbstractSecurityInterceptor` and is responsible for making final access control decisions. The `AccessDecisionManager` interface contains three methods:

```
void decide(Authentication authentication, Object secureObject,
    Collection<ConfigAttribute> attrs) throws AccessDeniedException;

boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

The `AccessDecisionManager`'s `decide` method is passed all the relevant information it needs in order to make an authorization decision. In particular, passing the secure `Object` enables those arguments contained in the actual secure object invocation to be inspected. For example, let's assume the secure object was a `MethodInvocation`. It would be easy to query the `MethodInvocation` for any `Customer` argument, and then implement some sort of security logic in the `AccessDecisionManager` to ensure the principal is permitted to operate on that customer. Implementations are expected to throw an `AccessDeniedException` if access is denied.

The `supports(ConfigAttribute)` method is called by the `AbstractSecurityInterceptor` at startup time to determine if the `AccessDecisionManager` can process the passed `ConfigAttribute`. The `supports(Class)` method is called by a security interceptor implementation to ensure the configured `AccessDecisionManager` supports the type of secure object that the security interceptor will present.

Voting-Based AccessDecisionManager Implementations

Whilst users can implement their own `AccessDecisionManager` to control all aspects of authorization, Spring Security includes several `AccessDecisionManager` implementations that are based on voting. Voting Decision Manager illustrates the relevant classes.

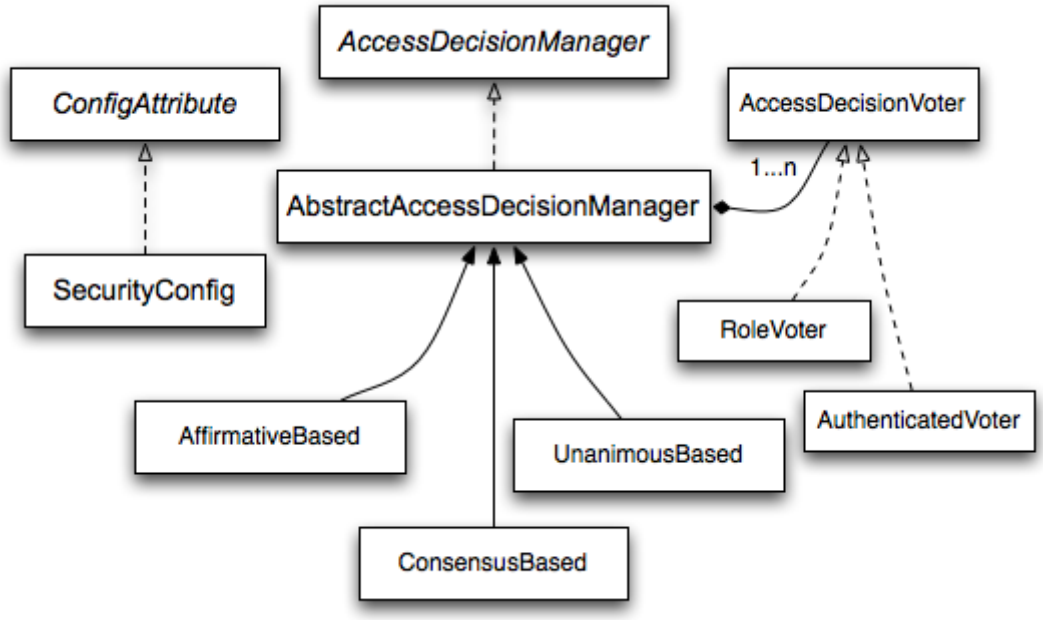


Figure 11. Voting Decision Manager

Using this approach, a series of AccessDecisionVoter implementations are polled on an authorization decision. The AccessDecisionManager then decides whether or not to throw an AccessDeniedException based on its assessment of the votes.

The AccessDecisionVoter interface has three methods:

```
int vote(Authentication authentication, Object object, Collection<ConfigAttribute>
attrs);

boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

Concrete implementations return an int, with possible values being reflected in the AccessDecisionVoter static fields ACCESS_ABSTAIN, ACCESS_DENIED and ACCESS_GRANTED. A voting implementation will return ACCESS_ABSTAIN if it has no opinion on an authorization decision. If it does have an opinion, it must return either ACCESS_DENIED or ACCESS_GRANTED.

There are three concrete AccessDecisionManager s provided with Spring Security that tally the votes. The ConsensusBased implementation will grant or deny access based on the consensus of non-abstain votes. Properties are provided to control behavior in the event of an equality of votes or if all votes are abstain. The AffirmativeBased implementation will grant access if one or more ACCESS_GRANTED votes were received (i.e. a deny vote will be ignored, provided there was at least one grant vote). Like the ConsensusBased implementation, there is a parameter that controls the behavior if all voters abstain. The UnanimousBased provider expects unanimous ACCESS_GRANTED votes in order to grant access, ignoring abstains. It will deny access if there is any ACCESS_DENIED vote. Like the other implementations, there is a parameter that controls the behaviour if all voters abstain.

It is possible to implement a custom AccessDecisionManager that tallies votes differently. For example, votes from a particular AccessDecisionVoter might receive additional weighting, whilst a deny vote from a particular voter may have a veto effect.

RoleVoter

The most commonly used AccessDecisionVoter provided with Spring Security is the simple RoleVoter, which treats configuration attributes as simple role names and votes to grant access if the user has been assigned that role.

It will vote if any ConfigAttribute begins with the prefix ROLE_. It will vote to grant access if there is a GrantedAuthority which returns a String representation (via the getAuthority() method) exactly equal to one or more ConfigAttributes starting with the prefix ROLE_. If there is no exact match of any ConfigAttribute starting with ROLE_, the RoleVoter will vote to deny access. If no ConfigAttribute begins with ROLE_, the voter will abstain.

AuthenticatedVoter

Another voter which we've implicitly seen is the AuthenticatedVoter, which can be used to differentiate between anonymous, fully-authenticated and remember-me authenticated users. Many sites allow certain limited access under remember-me authentication, but require a user to

confirm their identity by logging in for full access.

When we've used the attribute `IS_AUTHENTICATED_ANONYMOUSLY` to grant anonymous access, this attribute was being processed by the `AuthenticatedVoter`. See the Javadoc for this class for more information.

Custom Voters

Obviously, you can also implement a custom `AccessDecisionVoter` and you can put just about any access-control logic you want in it. It might be specific to your application (business-logic related) or it might implement some security administration logic. For example, you'll find a blog article on the Spring web site which describes how to use a voter to deny access in real-time to users whose accounts have been suspended.

11.1.3. After Invocation Handling

Whilst the `AccessDecisionManager` is called by the `AbstractSecurityInterceptor` before proceeding with the secure object invocation, some applications need a way of modifying the object actually returned by the secure object invocation. Whilst you could easily implement your own AOP concern to achieve this, Spring Security provides a convenient hook that has several concrete implementations that integrate with its ACL capabilities.

After Invocation Implementation illustrates Spring Security's `AfterInvocationManager` and its concrete implementations.

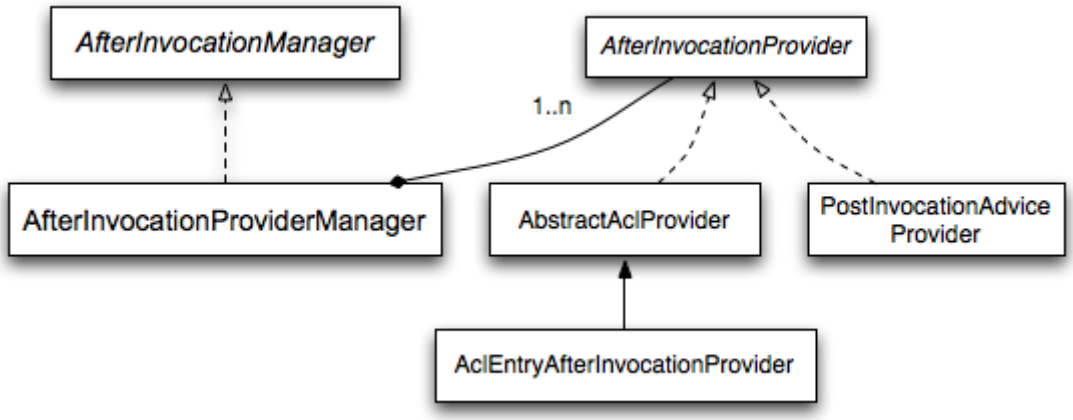


Figure 12. After Invocation Implementation

Like many other parts of Spring Security, `AfterInvocationManager` has a single concrete implementation, `AfterInvocationProviderManager`, which polls a list of `AfterInvocationProvider` s. Each `AfterInvocationProvider` is allowed to modify the return object or throw an `AccessDeniedException`. Indeed multiple providers can modify the object, as the result of the previous provider is passed to the next in the list.

Please be aware that if you're using `AfterInvocationManager`, you will still need configuration attributes that allow the `MethodSecurityInterceptor`'s `AccessDecisionManager` to allow an operation. If you're using the typical Spring Security included `AccessDecisionManager` implementations, having no configuration attributes defined for a particular secure method invocation will cause each `AccessDecisionVoter` to abstain from voting. In turn, if the `AccessDecisionManager` property

“allowIfAllAbstainDecisions” is `false`, an `AccessDeniedException` will be thrown. You may avoid this potential issue by either (i) setting “allowIfAllAbstainDecisions” to `true` (although this is generally not recommended) or (ii) simply ensure that there is at least one configuration attribute that an `AccessDecisionVoter` will vote to grant access for. This latter (recommended) approach is usually achieved through a `ROLE_USER` or `ROLE_AUTHENTICATED` configuration attribute.

11.1.4. Hierarchical Roles

It is a common requirement that a particular role in an application should automatically "include" other roles. For example, in an application which has the concept of an "admin" and a "user" role, you may want an admin to be able to do everything a normal user can. To achieve this, you can either make sure that all admin users are also assigned the "user" role. Alternatively, you can modify every access constraint which requires the "user" role to also include the "admin" role. This can get quite complicated if you have a lot of different roles in your application.

The use of a role-hierarchy allows you to configure which roles (or authorities) should include others. An extended version of Spring Security’s RoleVoter, `RoleHierarchyVoter`, is configured with a `RoleHierarchy`, from which it obtains all the "reachable authorities" which the user is assigned. A typical configuration might look like this:

```
<bean id="roleVoter"
class="org.springframework.security.access.vote.RoleHierarchyVoter">
    <constructor-arg ref="roleHierarchy" />
</bean>
<bean id="roleHierarchy"

class="org.springframework.security.access.hierarchicalroles.RoleHierarchyImpl">
    <property name="hierarchy">
        <value>
            ROLE_ADMIN > ROLE_STAFF
            ROLE_STAFF > ROLE_USER
            ROLE_USER > ROLE_GUEST
        </value>
    </property>
</bean>
```

Here we have four roles in a hierarchy `ROLE_ADMIN` ⇒ `ROLE_STAFF` ⇒ `ROLE_USER` ⇒ `ROLE_GUEST`. A user who is authenticated with `ROLE_ADMIN`, will behave as if they have all four roles when security constraints are evaluated against an `AccessDecisionManager` configured with the above `RoleHierarchyVoter`. The `>` symbol can be thought of as meaning "includes".

Role hierarchies offer a convenient means of simplifying the access-control configuration data for your application and/or reducing the number of authorities which you need to assign to a user. For more complex requirements you may wish to define a logical mapping between the specific access-rights your application requires and the roles that are assigned to users, translating between the two when loading the user information.

11.2. Authorize HttpServletRequest with FilterSecurityInterceptor

This section builds on Servlet Architecture and Implementation by digging deeper into how authorization works within Servlet based applications.

The `FilterSecurityInterceptor` provides authorization for `HttpServletRequest`s. It is inserted into the FilterChainProxy as one of the Security Filters.

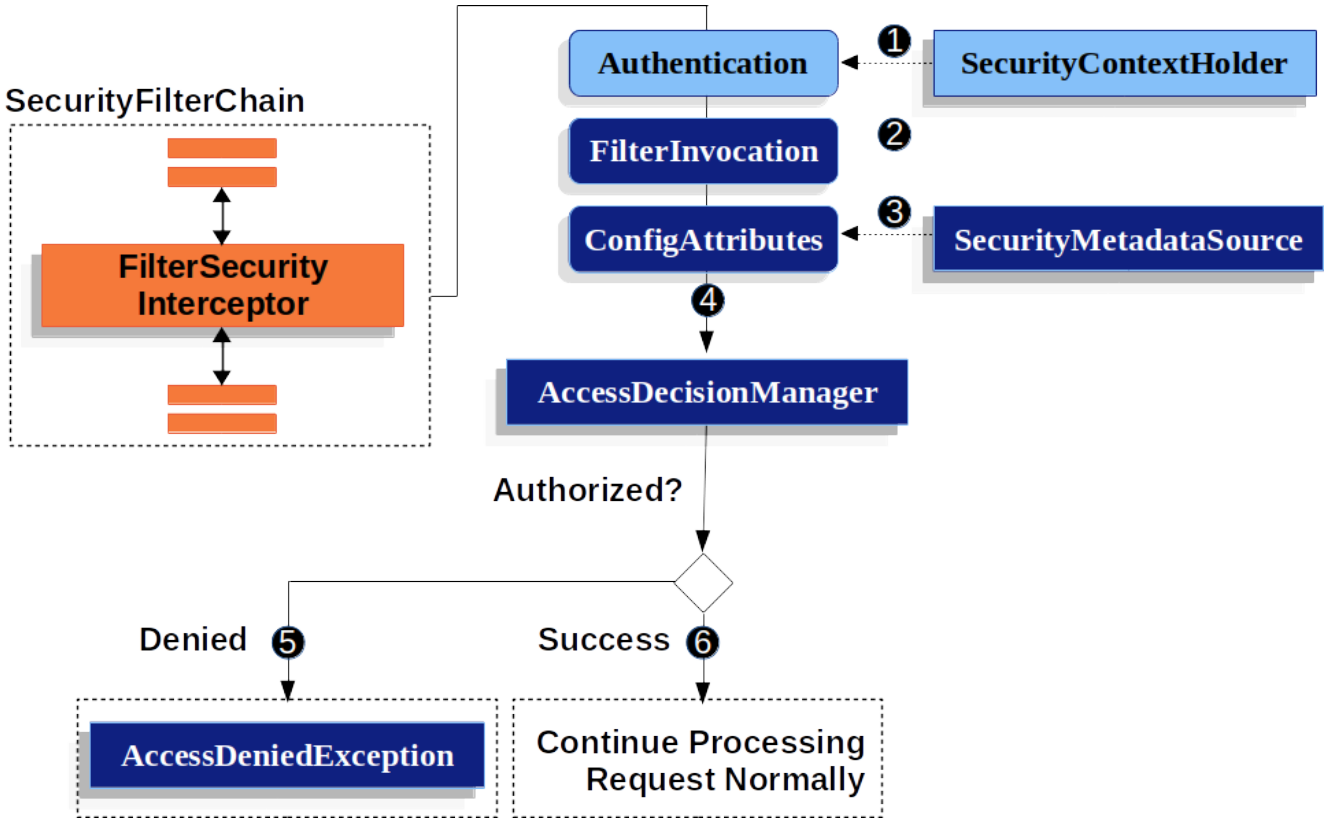


Figure 13. Authorize HttpServletRequest

- First, the `FilterSecurityInterceptor` obtains an Authentication from the SecurityContextHolder.
- Second, `FilterSecurityInterceptor` creates a `FilterInvocation` from the `HttpServletRequest`, `HttpServletResponse`, and `FilterChain` that are passed into the `FilterSecurityInterceptor`.
- Next, it passes the `FilterInvocation` to `SecurityMetadataSource` to get the `ConfigAttributes`.
- Finally, it passes the `Authentication`, `FilterInvocation`, and `ConfigAttributes` to the `AccessDecisionManager`.
  - If authorization is denied, an `AccessDeniedException` is thrown. In this case the `ExceptionTranslationFilter` handles the `AccessDeniedException`.
  - If access is granted, `FilterSecurityInterceptor` continues with the FilterChain which allows the application to process normally.

By default, Spring Security's authorization will require all requests to be authenticated. The explicit configuration looks like:

Example 88. Every Request Must be Authenticated

Java

```
protected void configure(HttpSecurity http) throws Exception {
    http
        // ...
        .authorizeRequests(authorize -> authorize
            .anyRequest().authenticated()
        );
}
```

XML

```
<http>
    <!-- ... -->
    <intercept-url pattern="/**" access="authenticated"/>
</http>
```

Kotlin

```
fun configure(http: HttpSecurity) {
    http {
        // ...
        authorizeRequests {
            authorize(anyRequest, authenticated)
        }
    }
}
```

We can configure Spring Security to have different rules by adding more rules in order of precedence.

Example 89. Authorize Requests

Java

```
protected void configure(HttpSecurity http) throws Exception {
    http
        // ...
        .authorizeRequests(authorize -> authorize
①
            .mvcMatchers("/resources/**", "/signup", "/about").permitAll()
②
            .mvcMatchers("/admin/**").hasRole("ADMIN")
③
            .mvcMatchers("/db/**").access("hasRole('ADMIN') and hasRole('DBA')")
④
            .anyRequest().denyAll()
⑤
        );
}
```

XML

```
<http> ①
    <!-- ... -->
    ②
    <intercept-url pattern="/resources/**" access="permitAll"/>
    <intercept-url pattern="/signup" access="permitAll"/>
    <intercept-url pattern="/about" access="permitAll"/>

    <intercept-url pattern="/admin/**" access="hasRole('ADMIN')"/> ③
    <intercept-url pattern="/db/**" access="hasRole('ADMIN') and hasRole('DBA')"/>
④
    <intercept-url pattern="/**" access="denyAll"/> ⑤
</http>
```

Kotlin

```
fun configure(http: HttpSecurity) {
   http {
        authorizeRequests { ①
            authorize("/resources/**", permitAll) ②
            authorize("/signup", permitAll)
            authorize("/about", permitAll)

            authorize("/admin/**", hasRole("ADMIN")) ③
            authorize("/db/**", "hasRole('ADMIN') and hasRole('DBA')") ④
            authorize(anyRequest, denyAll) ⑤
        }
    }
}
```

① There are multiple authorization rules specified. Each rule is considered in the order they were declared.

② We specified multiple URL patterns that any user can access. Specifically, any user can access a request if the URL starts with "/resources/", equals "/signup", or equals "/about".

③ Any URL that starts with "/admin/" will be restricted to users who have the role "ROLE_ADMIN". You will notice that since we are invoking the `hasRole` method we do not need to specify the "ROLE_" prefix.

④ Any URL that starts with "/db/" requires the user to have both "ROLE_ADMIN" and "ROLE_DBA". You will notice that since we are using the `hasRole` expression we do not need to specify the "ROLE_" prefix.

⑤ Any URL that has not already been matched on is denied access. This is a good strategy if you do not want to accidentally forget to update your authorization rules.

11.3. Expression-Based Access Control

Spring Security 3.0 introduced the ability to use Spring EL expressions as an authorization mechanism in addition to the simple use of configuration attributes and access-decision voters which have seen before. Expression-based access control is built on the same architecture but allows complicated Boolean logic to be encapsulated in a single expression.

11.3.1. Overview

Spring Security uses Spring EL for expression support and you should look at how that works if you are interested in understanding the topic in more depth. Expressions are evaluated with a "root object" as part of the evaluation context. Spring Security uses specific classes for web and method security as the root object, in order to provide built-in expressions and access to values such as the current principal.

Common Built-In Expressions

The base class for expression root objects is `SecurityExpressionRoot`. This provides some common expressions which are available in both web and method security.

Table 10. Common built-in expressions

Expression	Description
`hasRole(String role)`	Returns `true` if the current principal has the specified role. For example, `hasRole('admin')` By default if the supplied role does not start with 'ROLE_' it will be added. This can be customized by modifying the `defaultRolePrefix` on `DefaultWebSecurityExpressionHandler`.
`hasAnyRole(String… roles)`	Returns `true` if the current principal has any of the supplied roles (given as a comma-separated list of strings). For example, `hasAnyRole('admin', 'user')` By default if the supplied role does not start with 'ROLE_' it will be added. This can be customized by modifying the `defaultRolePrefix` on `DefaultWebSecurityExpressionHandler`.
`hasAuthority(String authority)`	Returns `true` if the current principal has the specified authority. For example, `hasAuthority('read')`
`hasAnyAuthority(String… authorities)`	Returns `true` if the current principal has any of the supplied authorities (given as a comma-separated list of strings) For example, `hasAnyAuthority('read', 'write')`
`principal`	Allows direct access to the principal object representing the current user
`authentication`	Allows direct access to the current `Authentication` object obtained from the `SecurityContext`
`permitAll`	Always evaluates to `true`
`denyAll`	Always evaluates to `false`
`isAnonymous()`	Returns `true` if the current principal is an anonymous user

Expression	Description
`isRememberMe()`	Returns `true` if the current principal is a remember-me user
`isAuthenticated()`	Returns `true` if the user is not anonymous
`isFullyAuthenticated()`	Returns `true` if the user is not an anonymous or a remember-me user
`hasPermission(Object target, Object permission)`	Returns `true` if the user has access to the provided target for the given permission. For example, `hasPermission(domainObject, 'read')`
`hasPermission(Object targetId, String targetType, Object permission)`	Returns `true` if the user has access to the provided target for the given permission. For example, `hasPermission(1, 'com.example.domain.Message', 'read')`

11.3.2. Web Security Expressions

To use expressions to secure individual URLs, you would first need to set the `use-expressions` attribute in the `<http>` element to `true`. Spring Security will then expect the `access` attributes of the `<intercept-url>` elements to contain Spring EL expressions. The expressions should evaluate to a Boolean, defining whether access should be allowed or not. For example:

```
<http>
    <intercept-url pattern="/admin*"
        access="hasRole('admin') and hasIpAddress('192.168.1.0/24')"/>
    ...
</http>
```

Here we have defined that the "admin" area of an application (defined by the URL pattern) should only be available to users who have the granted authority "admin" and whose IP address matches a local subnet. We've already seen the built-in `hasRole` expression in the previous section. The expression `hasIpAddress` is an additional built-in expression which is specific to web security. It is defined by the `WebSecurityExpressionRoot` class, an instance of which is used as the expression root object when evaluating web-access expressions. This object also directly exposed the `HttpServletRequest` object under the name `request` so you can invoke the request directly in an expression. If expressions are being used, a `WebExpressionVoter` will be added to the `AccessDecisionManager` which is used by the namespace. So if you aren't using the namespace and want to use expressions, you will have to add one of these to your configuration.

Referring to Beans in Web Security Expressions

If you wish to extend the expressions that are available, you can easily refer to any Spring Bean you expose. For example, assuming you have a Bean with the name of `webSecurity` that contains the following method signature:

Java

```
public class WebSecurity {
        public boolean check(Authentication authentication, HttpServletRequest
request) {
                ...
        }
}
```

Kotlin

```
class WebSecurity {
    fun check(authentication: Authentication?, request: HttpServletRequest?):
Boolean {
        // ...
    }
}
```

You could refer to the method using:

Example 90. Refer to method

Java

```
http
    .authorizeRequests(authorize -> authorize

.antMatchers("/user/**").access("@webSecurity.check(authentication,request)")
        ...
    )
```

XML

```
<http>
    <intercept-url pattern="/user/**"
        access="@webSecurity.check(authentication,request)"/>
    ...
</http>
```

Kotlin

```
http {
    authorizeRequests {
        authorize("/user/**", "@webSecurity.check(authentication,request)")
    }
}
```

Path Variables in Web Security Expressions

At times it is nice to be able to refer to path variables within a URL. For example, consider a RESTful application that looks up a user by id from the URL path in the format `/user/{userId}`.

You can easily refer to the path variable by placing it in the pattern. For example, if you had a Bean with the name of `webSecurity` that contains the following method signature:

Java

```
public class WebSecurity {
        public boolean checkUserId(Authentication authentication, int id) {
                ...
        }
}
```

Kotlin

```
class WebSecurity {
    fun checkUserId(authentication: Authentication?, id: Int): Boolean {
        // ...
    }
}
```

You could refer to the method using:

Example 91. Path Variables

Java

```
http
    .authorizeRequests(authorize -> authorize

.antMatchers("/user/{userId}/**").access("@webSecurity.checkUserId(authentication,
#userId)")
        ...
    );
```

XML

```
<http>
    <intercept-url pattern="/user/{userId}/**"
        access="@webSecurity.checkUserId(authentication,#userId)"/>
    ...
</http>
```

Kotlin

```
http {
    authorizeRequests {
        authorize("/user/{userId}/**",
"@webSecurity.checkUserId(authentication,#userId)")
    }
}
```

In this configuration URLs that match would pass in the path variable (and convert it) into checkUserId method. For example, if the URL were `/user/123/resource`, then the id passed in would be `123`.

11.3.3. Method Security Expressions

Method security is a bit more complicated than a simple allow or deny rule. Spring Security 3.0 introduced some new annotations in order to allow comprehensive support for the use of expressions.

@Pre and @Post Annotations

There are four annotations which support expression attributes to allow pre and post-invocation authorization checks and also to support filtering of submitted collection arguments or return values. They are `@PreAuthorize`, `@PreFilter`, `@PostAuthorize` and `@PostFilter`. Their use is enabled through the `global-method-security` namespace element:

```
<global-method-security pre-post-annotations="enabled"/>
```

Access Control using @PreAuthorize and @PostAuthorize

The most obviously useful annotation is `@PreAuthorize` which decides whether a method can actually be invoked or not. For example (from the "Contacts" sample application)

Java

```
@PreAuthorize("hasRole('USER')")
public void create(Contact contact);
```

Kotlin

```
@PreAuthorize("hasRole('USER')")
fun create(contact: Contact?)
```

which means that access will only be allowed for users with the role "ROLE_USER". Obviously the same thing could easily be achieved using a traditional configuration and a simple configuration attribute for the required role. But what about:

Java

```
@PreAuthorize("hasPermission(#contact, 'admin')")
public void deletePermission(Contact contact, Sid recipient, Permission
permission);
```

Kotlin

```
@PreAuthorize("hasPermission(#contact, 'admin')")
fun deletePermission(contact: Contact?, recipient: Sid?, permission: Permission?)
```

Here we're actually using a method argument as part of the expression to decide whether the current user has the "admin"permission for the given contact. The built-in `hasPermission()` expression is linked into the Spring Security ACL module through the application context, as we'll see below. You can access any of the method arguments by name as expression variables.

There are a number of ways in which Spring Security can resolve the method arguments. Spring Security uses `DefaultSecurityParameterNameDiscoverer` to discover the parameter names. By default, the following options are tried for a method as a whole.

- If Spring Security's `@P` annotation is present on a single argument to the method, the value will be used. This is useful for interfaces compiled with a JDK prior to JDK 8 which do not contain any information about the parameter names. For example:

Java

```
import org.springframework.security.access.method.P;

...

@PreAuthorize("#c.name == authentication.name")
public void doSomething(@P("c") Contact contact);
```

Kotlin

```
import org.springframework.security.access.method.P

...

@PreAuthorize("#c.name == authentication.name")
fun doSomething(@P("c") contact: Contact?)
```

Behind the scenes this is implemented using `AnnotationParameterNameDiscoverer` which can be customized to support the value attribute of any specified annotation.

- If Spring Data's `@Param` annotation is present on at least one parameter for the method, the value will be used. This is useful for interfaces compiled with a JDK prior to JDK 8 which do not contain any information about the parameter names. For example:

Java

```
import org.springframework.data.repository.query.Param;

...

@PreAuthorize("#n == authentication.name")
Contact findContactByName(@Param("n") String name);
```

Kotlin

```
import org.springframework.data.repository.query.Param

...

@PreAuthorize("#n == authentication.name")
fun findContactByName(@Param("n") name: String?): Contact?
```

Behind the scenes this is implemented using `AnnotationParameterNameDiscoverer` which can be customized to support the value attribute of any specified annotation.

- If JDK 8 was used to compile the source with the -parameters argument and Spring 4+ is being used, then the standard JDK reflection API is used to discover the parameter names. This works on both classes and interfaces.
- Last, if the code was compiled with the debug symbols, the parameter names will be discovered using the debug symbols. This will not work for interfaces since they do not have debug information about the parameter names. For interfaces, annotations or the JDK 8 approach must be used.

Any Spring-EL functionality is available within the expression, so you can also access properties on the arguments. For example, if you wanted a particular method to only allow access to a user whose username matched that of the contact, you could write

Java

```
@PreAuthorize("#contact.name == authentication.name")
public void doSomething(Contact contact);
```

Kotlin

```
@PreAuthorize("#contact.name == authentication.name")
fun doSomething(contact: Contact?)
```

Here we are accessing another built-in expression, `authentication`, which is the `Authentication` stored in the security context. You can also access its "principal" property directly, using the expression `principal`. The value will often be a `UserDetails` instance, so you might use an expression like `principal.username` or `principal.enabled`.

Less commonly, you may wish to perform an access-control check after the method has been invoked. This can be achieved using the `@PostAuthorize` annotation. To access the return value from a method, use the built-in name `returnObject` in the expression.

Filtering using @PreFilter and @PostFilter

Spring Security supports filtering of collections, arrays, maps and streams using expressions. This is most commonly performed on the return value of a method. For example:

Java

```
@PreAuthorize("hasRole('USER')")
@PostFilter("hasPermission(filterObject, 'read') or hasPermission(filterObject,
'admin')")
public List<Contact> getAll();
```

Kotlin

```
@PreAuthorize("hasRole('USER')")
@PostFilter("hasPermission(filterObject, 'read') or hasPermission(filterObject,
'admin')")
fun getAll(): List<Contact?>
```

When using the `@PostFilter` annotation, Spring Security iterates through the returned collection or map and removes any elements for which the supplied expression is false. For an array, a new array instance will be returned containing filtered elements. The name `filterObject` refers to the current object in the collection. In case when a map is used it will refer to the current `Map.Entry` object which allows one to use `filterObject.key` or `filterObject.value` in the expresion. You can also filter before the method call, using `@PreFilter`, though this is a less common requirement. The syntax is just the same, but if there is more than one argument which is a collection type then you have to select one by name using the `filterTarget` property of this annotation.

Note that filtering is obviously not a substitute for tuning your data retrieval queries. If you are filtering large collections and removing many of the entries then this is likely to be inefficient.

Built-In Expressions

There are some built-in expressions which are specific to method security, which we have already seen in use above. The `filterTarget` and `returnValue` values are simple enough, but the use of the `hasPermission()` expression warrants a closer look.

The PermissionEvaluator interface

`hasPermission()` expressions are delegated to an instance of `PermissionEvaluator`. It is intended to bridge between the expression system and Spring Security's ACL system, allowing you to specify authorization constraints on domain objects, based on abstract permissions. It has no explicit dependencies on the ACL module, so you could swap that out for an alternative implementation if required. The interface has two methods:

```
boolean hasPermission(Authentication authentication, Object targetDomainObject,
                            Object permission);

boolean hasPermission(Authentication authentication, Serializable targetId,
                            String targetType, Object permission);
```

which map directly to the available versions of the expression, with the exception that the first argument (the `Authentication` object) is not supplied. The first is used in situations where the domain object, to which access is being controlled, is already loaded. Then expression will return true if the current user has the given permission for that object. The second version is used in cases where the object is not loaded, but its identifier is known. An abstract "type" specifier for the domain object is also required, allowing the correct ACL permissions to be loaded. This has traditionally been the Java class of the object, but does not have to be as long as it is consistent with how the permissions are loaded.

To use `hasPermission()` expressions, you have to explicitly configure a `PermissionEvaluator` in your application context. This would look something like this:

```
<security:global-method-security pre-post-annotations="enabled">
<security:expression-handler ref="expressionHandler"/>
</security:global-method-security>

<bean id="expressionHandler" class=
"org.springframework.security.access.expression.method.DefaultMethodSecurityExpression
Handler">
    <property name="permissionEvaluator" ref="myPermissionEvaluator"/>
</bean>
```

Where `myPermissionEvaluator` is the bean which implements `PermissionEvaluator`. Usually this will be the implementation from the ACL module which is called `AclPermissionEvaluator`. See the "Contacts" sample application configuration for more details.

Method Security Meta Annotations

You can make use of meta annotations for method security to make your code more readable. This is especially convenient if you find that you are repeating the same complex expression throughout your code base. For example, consider the following:

```
@PreAuthorize("#contact.name == authentication.name")
```

Instead of repeating this everywhere, we can create a meta annotation that can be used instead.

Java

```
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("#contact.name == authentication.name")
public @interface ContactPermission {}
```

Kotlin

```
@Retention(AnnotationRetention.RUNTIME)
@PreAuthorize("#contact.name == authentication.name")
annotation class ContactPermission
```

Meta annotations can be used for any of the Spring Security method security annotations. In order to remain compliant with the specification JSR-250 annotations do not support meta annotations.

11.4. Secure Object Implementations

11.4.1. AOP Alliance (MethodInvocation) Security Interceptor

Prior to Spring Security 2.0, securing `MethodInvocation` s needed quite a lot of boiler plate configuration. Now the recommended approach for method security is to use namespace configuration. This way the method security infrastructure beans are configured automatically for you so you don't really need to know about the implementation classes. We'll just provide a quick overview of the classes that are involved here.

Method security is enforced using a `MethodSecurityInterceptor`, which secures `MethodInvocation` s. Depending on the configuration approach, an interceptor may be specific to a single bean or shared between multiple beans. The interceptor uses a `MethodSecurityMetadataSource` instance to obtain the configuration attributes that apply to a particular method invocation. `MapBasedMethodSecurityMetadataSource` is used to store configuration attributes keyed by method names (which can be wildcarded) and will be used internally when the attributes are defined in the application context using the `<intercept-methods>` or `<protect-point>` elements. Other implementations will be used to handle annotation-based configuration.

Explicit MethodSecurityInterceptor Configuration

You can of course configure a `MethodSecurityInterceptor` directly in your application context for use with one of Spring AOP's proxying mechanisms:

```
<bean id="bankManagerSecurity" class=

"org.springframework.security.access.intercept.aopalliance.MethodSecurityInterceptor">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="accessDecisionManager"/>
<property name="afterInvocationManager" ref="afterInvocationManager"/>
<property name="securityMetadataSource">
    <sec:method-security-metadata-source>
    <sec:protect method="com.mycompany.BankManager.delete*" access="ROLE_SUPERVISOR"/>
    <sec:protect method="com.mycompany.BankManager.getBalance"
access="ROLE_TELLER,ROLE_SUPERVISOR"/>
    </sec:method-security-metadata-source>
</property>
</bean>
```

11.4.2. AspectJ (JoinPoint) Security Interceptor

The AspectJ security interceptor is very similar to the AOP Alliance security interceptor discussed in the previous section. Indeed we will only discuss the differences in this section.

The AspectJ interceptor is named `AspectJSecurityInterceptor`. Unlike the AOP Alliance security interceptor, which relies on the Spring application context to weave in the security interceptor via proxying, the `AspectJSecurityInterceptor` is weaved in via the AspectJ compiler. It would not be uncommon to use both types of security interceptors in the same application, with `AspectJSecurityInterceptor` being used for domain object instance security and the AOP Alliance `MethodSecurityInterceptor` being used for services layer security.

Let's first consider how the `AspectJSecurityInterceptor` is configured in the Spring application context:

```
<bean id="bankManagerSecurity" class=

"org.springframework.security.access.intercept.aspectj.AspectJMethodSecurityIntercepto
r">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="accessDecisionManager"/>
<property name="afterInvocationManager" ref="afterInvocationManager"/>
<property name="securityMetadataSource">
    <sec:method-security-metadata-source>
    <sec:protect method="com.mycompany.BankManager.delete*" access="ROLE_SUPERVISOR"/>
    <sec:protect method="com.mycompany.BankManager.getBalance"
access="ROLE_TELLER,ROLE_SUPERVISOR"/>
    </sec:method-security-metadata-source>
</property>
</bean>
```

As you can see, aside from the class name, the `AspectJSecurityInterceptor` is exactly the same as the AOP Alliance security interceptor. Indeed the two interceptors can share the same

`securityMetadataSource`, as the `SecurityMetadataSource` works with `java.lang.reflect.Method` s rather than an AOP library-specific class. Of course, your access decisions have access to the relevant AOP library-specific invocation (ie `MethodInvocation` or `JoinPoint`) and as such can consider a range of addition criteria when making access decisions (such as method arguments).

Next you'll need to define an AspectJ `aspect`. For example:

```
package org.springframework.security.samples.aspectj;

import
org.springframework.security.access.intercept.aspectj.AspectJSecurityInterceptor;
import org.springframework.security.access.intercept.aspectj.AspectJCallback;
import org.springframework.beans.factory.InitializingBean;

public aspect DomainObjectInstanceSecurityAspect implements InitializingBean {

    private AspectJSecurityInterceptor securityInterceptor;

    pointcut domainObjectInstanceExecution(): target(PersistableEntity)
        && execution(public * *(..)) && !within(DomainObjectInstanceSecurityAspect);

    Object around(): domainObjectInstanceExecution() {
        if (this.securityInterceptor == null) {
            return proceed();
        }

        AspectJCallback callback = new AspectJCallback() {
            public Object proceedWithObject() {
                return proceed();
            }
        };

        return this.securityInterceptor.invoke(thisJoinPoint, callback);
    }

    public AspectJSecurityInterceptor getSecurityInterceptor() {
        return securityInterceptor;
    }

    public void setSecurityInterceptor(AspectJSecurityInterceptor securityInterceptor)
{
        this.securityInterceptor = securityInterceptor;
    }

    public void afterPropertiesSet() throws Exception {
        if (this.securityInterceptor == null)
            throw new IllegalArgumentException("securityInterceptor required");
        }
    }
}
```

In the above example, the security interceptor will be applied to every instance of `PersistableEntity`, which is an abstract class not shown (you can use any other class or `pointcut` expression you like). For those curious, `AspectJCallback` is needed because the `proceed();` statement has special meaning only within an `around()` body. The `AspectJSecurityInterceptor` calls this anonymous `AspectJCallback` class when it wants the target object to continue.

You will need to configure Spring to load the aspect and wire it with the `AspectJSecurityInterceptor`. A bean declaration which achieves this is shown below:

```
<bean id="domainObjectInstanceSecurityAspect"
    class="security.samples.aspectj.DomainObjectInstanceSecurityAspect"
    factory-method="aspectOf">
<property name="securityInterceptor" ref="bankManagerSecurity"/>
</bean>
```

That's it! Now you can create your beans from anywhere within your application, using whatever means you think fit (e.g. `new Person();`) and they will have the security interceptor applied.

11.5. Method Security

From version 2.0 onwards Spring Security has improved support substantially for adding security to your service layer methods. It provides support for JSR-250 annotation security as well as the framework's original `@Secured` annotation. From 3.0 you can also make use of new expression-based annotations. You can apply security to a single bean, using the `intercept-methods` element to decorate the bean declaration, or you can secure multiple beans across the entire service layer using the AspectJ style pointcuts.

11.5.1. EnableMethodSecurity

In Spring Security 5.6, we can enable annotation-based security using the `@EnableMethodSecurity` annotation on any `@Configuration` instance.

This improves upon `@EnableGlobalMethodSecurity` in a number of ways. `@EnableMethodSecurity`:

1. Uses the simplified `AuthorizationManager` API instead of metadata sources, config attributes, decision managers, and voters. This simplifies reuse and customization.
2. Favors direct bean-based configuration, instead of requiring extending `GlobalMethodSecurityConfiguration` to customize beans
3. Is built using native Spring AOP, removing abstractions and allowing you to use Spring AOP building blocks to customize
4. Checks for conflicting annotations to ensure an unambiguous security configuration
5. Complies with JSR-250
6. Enables `@PreAuthorize`, `@PostAuthorize`, `@PreFilter`, and `@PostFilter` by default

For earlier versions, please read about similar support with @EnableGlobalMethodSecurity.

For example, the following would enable Spring Security's `@PreAuthorize` annotation:

Example 92. Method Security Configuration

Java

```
@EnableMethodSecurity
public class MethodSecurityConfig {
    // ...
}
```

Kotlin

```
@EnableMethodSecurity
class MethodSecurityConfig {
    // ...
}
```

Xml

```
<sec:method-security/>
```

Adding an annotation to a method (on a class or interface) would then limit the access to that method accordingly. Spring Security's native annotation support defines a set of attributes for the method. These will be passed to the `DefaultAuthorizationMethodInterceptorChain` for it to make the actual decision:

Example 93. Method Security Annotation Usage

Java

```
public interface BankService {
    @PreAuthorize("hasRole('USER')")
    Account readAccount(Long id);

    @PreAuthorize("hasRole('USER')")
    List<Account> findAccounts();

    @PreAuthorize("hasRole('TELLER')")
    Account post(Account account, Double amount);
}
```

Kotlin

```
interface BankService {
    @PreAuthorize("hasRole('USER')")
    fun readAccount(id : Long) : Account

    @PreAuthorize("hasRole('USER')")
    fun findAccounts() : List<Account>

    @PreAuthorize("hasRole('TELLER')")
    fun post(account : Account, amount : Double) : Account
}
```

You can enable support for Spring Security's `@Secured` annotation using:

Example 94. @Secured Configuration

Java

```
@EnableMethodSecurity(securedEnabled = true)
public class MethodSecurityConfig {
    // ...
}
```

Kotlin

```
@EnableMethodSecurity(securedEnabled = true)
class MethodSecurityConfig {
    // ...
}
```

Xml

```
<sec:method-security secured-enabled="true"/>
```

or JSR-250 using:

Example 95. JSR-250 Configuration

Java

```
@EnableMethodSecurity(jsr250Enabled = true)
public class MethodSecurityConfig {
    // ...
}
```

Kotlin

```
@EnableMethodSecurity(jsr250Enabled = true)
class MethodSecurityConfig {
    // ...
}
```

Xml

```
<sec:method-security jsr250-enabled="true"/>
```

Customizing Authorization

Spring Security's `@PreAuthorize`, `@PostAuthorize`, `@PreFilter`, and `@PostFilter` ship with rich

expression-based support.

If you need to customize the way that expressions are handled, you can expose a custom `MethodSecurityExpressionHandler`, like so:

Example 96. Custom MethodSecurityExpressionHandler

Java

```
@Bean
static MethodSecurityExpressionHandler methodSecurityExpressionHandler() {
    DefaultMethodSecurityExpressionHandler handler = new
DefaultMethodSecurityExpressionHandler();
    handler.setTrustResolver(myCustomTrustResolver);
    return handler;
}
```

Kotlin

```
companion object {
    @Bean
    fun methodSecurityExpressionHandler() : MethodSecurityExpressionHandler {
        val handler = DefaultMethodSecurityExpressionHandler();
        handler.setTrustResolver(myCustomTrustResolver);
        return handler;
    }
}
```

Xml

```
<sec:method-security>
    <sec:expression-handler ref="myExpressionHandler"/>
</sec:method-security>

<bean id="myExpressionHandler"

class="org.springframework.security.messaging.access.expression.DefaultMessageSecu
rityExpressionHandler">
    <property name="trustResolver" ref="myCustomTrustResolver"/>
</bean>
```

We expose `MethodSecurityExpressionHandler` using a `static` method to ensure that Spring publishes it before it initializes Spring Security's method security `@Configuration` classes

Also, for role-based authorization, Spring Security adds a default `ROLE_` prefix, which is uses when evaluating expressions like `hasRole`.

You can configure the authorization rules to use a different prefix by exposing a `GrantedAuthorityDefaults` bean, like so:

Example 97. Custom MethodSecurityExpressionHandler

Java

```
@Bean
static GrantedAuthorityDefaults grantedAuthorityDefaults() {
    return new GrantedAuthorityDefaults("MYPREFIX_");
}
```

Kotlin

```
companion object {
    @Bean
    fun grantedAuthorityDefaults() : GrantedAuthorityDefaults {
        return GrantedAuthorityDefaults("MYPREFIX_");
    }
}
```

Xml

```
<sec:method-security/>

<bean id="grantedAuthorityDefaults"
class="org.springframework.security.config.core.GrantedAuthorityDefaults">
    <constructor-arg value="MYPREFIX_"/>
</bean>
```

We expose `GrantedAuthorityDefaults` using a `static` method to ensure that Spring publishes it before it initializes Spring Security's method security `@Configuration` classes

Custom Authorization Managers

Method authorization is a combination of before- and after-method authorization.

Before-method authorization is performed before the method is invoked. If that authorization denies access, the method is not invoked, and an `AccessDeniedException` is thrown After-method authorization is performed after the method is invoked, but before the method returns to the caller. If that authorization denies access, the value is not returned, and an `AccessDeniedException` is thrown

To recreate what adding `@EnableMethodSecurity` does by default, you would publish the following configuration:

Example 98. Full Pre-post Method Security Configuration

Java

```
@EnableMethodSecurity(prePostEnabled = false)
class MethodSecurityConfig {
    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    Advisor preFilterAuthorizationMethodInterceptor() {
        return new PreFilterAuthorizationMethodInterceptor();
    }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    Advisor preAuthorizeAuthorizationMethodInterceptor() {
        return AuthorizationManagerBeforeMethodInterceptor.preAuthorize();
    }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    Advisor postAuthorizeAuthorizationMethodInterceptor() {
        return AuthorizationManagerAfterMethodInterceptor.postAuthorize();
    }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    Advisor postFilterAuthorizationMethodInterceptor() {
        return new PostFilterAuthorizationMethodInterceptor();
    }
}
```

Kotlin

```
@EnableMethodSecurity(prePostEnabled = false)
class MethodSecurityConfig {
    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    fun preFilterAuthorizationMethodInterceptor() : Advisor {
        return PreFilterAuthorizationMethodInterceptor();
    }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    fun preAuthorizeAuthorizationMethodInterceptor() : Advisor {
        return AuthorizationManagerBeforeMethodInterceptor.preAuthorize();
    }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    fun postAuthorizeAuthorizationMethodInterceptor() : Advisor {
        return AuthorizationManagerAfterMethodInterceptor.postAuthorize();
    }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    fun postFilterAuthorizationMethodInterceptor() : Advisor {
        return PostFilterAuthorizationMethodInterceptor();
    }
}
```

Xml

```
<sec:method-security pre-post-enabled="false"/>

<aop:config/>

<bean id="preFilterAuthorizationMethodInterceptor"

class="org.springframework.security.authorization.method.PreFilterAuthorizationMet
hodInterceptor"/>
<bean id="preAuthorizeAuthorizationMethodInterceptor"

class="org.springframework.security.authorization.method.AuthorizationManagerBefor
eMethodInterceptor"
        factory-method="preAuthorize"/>
<bean id="postAuthorizeAuthorizationMethodInterceptor"

class="org.springframework.security.authorization.method.AuthorizationManagerAfter
MethodInterceptor"
        factory-method="postAuthorize"/>
<bean id="postFilterAuthorizationMethodInterceptor"

class="org.springframework.security.authorization.method.PostFilterAuthorizationMe
thodInterceptor"/>
```

Notice that Spring Security's method security is built using Spring AOP. So, interceptors are invoked based on the order specified. This can be customized by calling `setOrder` on the interceptor instances like so:

Example 99. Publish Custom Advisor

Java

```
@Bean
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
Advisor postFilterAuthorizationMethodInterceptor() {
    PostFilterAuthorizationMethodInterceptor interceptor = new
PostFilterAuthorizationMethodInterceptor();
    interceptor.setOrder(AuthorizationInterceptorOrders.POST_AUTHORIZE.getOrder()
- 1);
    return interceptor;
}
```

Kotlin

```
@Bean
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
fun postFilterAuthorizationMethodInterceptor() : Advisor {
    val interceptor = PostFilterAuthorizationMethodInterceptor();
    interceptor.setOrder(AuthorizationInterceptorOrders.POST_AUTHORIZE.getOrder()
- 1);
    return interceptor;
}
```

Xml

```
<bean id="postFilterAuthorizationMethodInterceptor"

class="org.springframework.security.authorization.method.PostFilterAuthorizationMe
thodInterceptor">
    <property name="order"

value="#{T(org.springframework.security.authorization.method.AuthorizationIntercep
torsOrder).POST_AUTHORIZE.getOrder() -1}"/>
</bean>
```

You may want to only support `@PreAuthorize` in your application, in which case you can do the following:

Example 100. Only @PreAuthorize Configuration

Java

```
@EnableMethodSecurity(prePostEnabled = false)
class MethodSecurityConfig {
    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    Advisor preAuthorize() {
        return AuthorizationManagerBeforeMethodInterceptor.preAuthorize();
    }
}
```

Kotlin

```
@EnableMethodSecurity(prePostEnabled = false)
class MethodSecurityConfig {
    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    fun preAuthorize() : Advisor {
        return AuthorizationManagerBeforeMethodInterceptor.preAuthorize()
    }
}
```

Xml

```
<sec:method-security pre-post-enabled="false"/>

<aop:config/>

<bean id="preAuthorizeAuthorizationMethodInterceptor"

class="org.springframework.security.authorization.method.AuthorizationManagerBefor
eMethodInterceptor"
    factory-method="preAuthorize"/>
```

Or, you may have a custom before-method `AuthorizationManager` that you want to add to the list.

In this case, you will need to tell Spring Security both the `AuthorizationManager` and to which methods and classes your authorization manager applies.

Thus, you can configure Spring Security to invoke your `AuthorizationManager` in between `@PreAuthorize` and `@PostAuthorize` like so:

Example 101. Custom Before Advisor

Java

```
@EnableMethodSecurity
class MethodSecurityConfig {
    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public Advisor customAuthorize() {
        JdkRegexpMethodPointcut pattern = new JdkRegexpMethodPointcut();
        pattern.setPattern("org.mycompany.myapp.service.*");
        AuthorizationManager<MethodInvocation> rule =
AuthorityAuthorizationManager.isAuthenticated();
        AuthorizationManagerBeforeMethodInterceptor interceptor = new
AuthorizationManagerBeforeMethodInterceptor(pattern, rule);

interceptor.setOrder(AuthorizationInterceptorsOrder.PRE_AUTHORIZE_ADVISOR_ORDER.ge
tOrder() + 1);
        return interceptor;
    }
}
```

Kotlin

```
@EnableMethodSecurity
class MethodSecurityConfig {
    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    fun customAuthorize() : Advisor {
        val pattern = JdkRegexpMethodPointcut();
        pattern.setPattern("org.mycompany.myapp.service.*");
        val rule = AuthorityAuthorizationManager.isAuthenticated();
        val interceptor = AuthorizationManagerBeforeMethodInterceptor(pattern,
rule);

interceptor.setOrder(AuthorizationInterceptorsOrder.PRE_AUTHORIZE_ADVISOR_ORDER.ge
tOrder() + 1);
        return interceptor;
    }
}
```

Xml

```
<sec:method-security/>

<aop:config/>

<bean id="customAuthorize"

class="org.springframework.security.authorization.method.AuthorizationManagerBefor
eMethodInterceptor">
    <constructor-arg>
        <bean class="org.springframework.aop.support.JdkRegexpMethodPointcut">
            <property name="pattern" value="org.mycompany.myapp.service.*"/>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean
class="org.springframework.security.authorization.AuthorityAuthorizationManager"
                factory-method="isAuthenticated"/>
    </constructor-arg>
    <property name="order"

value="#{T(org.springframework.security.authorization.method.AuthorizationIntercep
torsOrder).PRE_AUTHORIZE_ADVISOR_ORDER.getOrder() + 1}"/>
</bean>
```

You can place your interceptor in between Spring Security method interceptors using the order constants specified in `AuthorizationInterceptorsOrder`.

The same can be done for after-method authorization. After-method authorization is generally concerned with analysing the return value to verify access.

For example, you might have a method that confirms that the account requested actually belongs to the logged-in user like so:

Example 102. @PostAuthorize example

Java

```
public interface BankService {

    @PreAuthorize("hasRole('USER')")
    @PostAuthorize("returnObject.owner == authentication.name")
    Account readAccount(Long id);
}
```

Kotlin

```
interface BankService {

    @PreAuthorize("hasRole('USER')")
    @PostAuthorize("returnObject.owner == authentication.name")
    fun readAccount(id : Long) : Account
}
```

You can supply your own `AuthorizationMethodInterceptor` to customize how access to the return value is evaluated.

For example, if you have your own custom annotation, you can configure it like so:

Example 103. Custom After Advisor

Java

```
@EnableMethodSecurity
class MethodSecurityConfig {
    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public Advisor customAuthorize(AuthorizationManager<MethodInvocationResult>
rules) {
        AnnotationMethodMatcher pattern = new
AnnotationMethodMatcher(MySecurityAnnotation.class);
        AuthorizationManagerAfterMethodInterceptor interceptor = new
AuthorizationManagerAfterMethodInterceptor(pattern, rules);

interceptor.setOrder(AuthorizationInterceptorsOrder.POST_AUTHORIZE_ADVISOR_ORDER.g
etOrder() + 1);
        return interceptor;
    }
}
```

Kotlin

```
@EnableMethodSecurity
class MethodSecurityConfig {
    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    fun customAuthorize(rules : AuthorizationManager<MethodInvocationResult>) :
Advisor {
        val pattern = AnnotationMethodMatcher(MySecurityAnnotation::class.java);
        val interceptor = AuthorizationManagerAfterMethodInterceptor(pattern,
rules);

interceptor.setOrder(AuthorizationInterceptorsOrder.POST_AUTHORIZE_ADVISOR_ORDER.g
etOrder() + 1);
        return interceptor;
    }
}
```

Xml

```
<sec:method-security/>

<aop:config/>

<bean id="customAuthorize"

class="org.springframework.security.authorization.method.AuthorizationManagerAfter
MethodInterceptor">
    <constructor-arg>
        <bean
class="org.springframework.aop.support.annotation.AnnotationMethodMatcher">
            <constructor-arg value="#{T(org.mycompany.MySecurityAnnotation)}"/>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean
class="org.springframework.security.authorization.AuthorityAuthorizationManager"
                factory-method="isAuthenticated"/>
    </constructor-arg>
    <property name="order"

value="#{T(org.springframework.security.authorization.method.AuthorizationIntercep
torsOrder).PRE_AUTHORIZE_ADVISOR_ORDER.getOrder() + 1}"/>
</bean>
```

and it will be invoked after the `@PostAuthorize` interceptor.

11.5.2. EnableGlobalMethodSecurity

We can enable annotation-based security using the `@EnableGlobalMethodSecurity` annotation on any `@Configuration` instance. For example, the following would enable Spring Security's `@Secured` annotation.

Java

```
@EnableGlobalMethodSecurity(securedEnabled = true)
public class MethodSecurityConfig {
// ...
}
```

Kotlin

```
@EnableGlobalMethodSecurity(securedEnabled = true)
open class MethodSecurityConfig {
    // ...
}
```

Adding an annotation to a method (on a class or interface) would then limit the access to that method accordingly. Spring Security's native annotation support defines a set of attributes for the method. These will be passed to the AccessDecisionManager for it to make the actual decision:

Java

```
public interface BankService {

@Secured("IS_AUTHENTICATED_ANONYMOUSLY")
public Account readAccount(Long id);

@Secured("IS_AUTHENTICATED_ANONYMOUSLY")
public Account[] findAccounts();

@Secured("ROLE_TELLER")
public Account post(Account account, double amount);
}
```

Kotlin

```
interface BankService {
    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    fun readAccount(id: Long): Account

    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    fun findAccounts(): Array<Account>

    @Secured("ROLE_TELLER")
    fun post(account: Account, amount: Double): Account
}
```

Support for JSR-250 annotations can be enabled using

Java

```
@EnableGlobalMethodSecurity(jsr250Enabled = true)
public class MethodSecurityConfig {
// ...
}
```

Kotlin

```
@EnableGlobalMethodSecurity(jsr250Enabled = true)
open class MethodSecurityConfig {
    // ...
}
```

These are standards-based and allow simple role-based constraints to be applied but do not have the power Spring Security's native annotations. To use the new expression-based syntax, you would use

Java

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig {
// ...
}
```

Kotlin

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
open class MethodSecurityConfig {
    // ...
}
```

and the equivalent Java code would be

Java

```
public interface BankService {

@PreAuthorize("isAnonymous()")
public Account readAccount(Long id);

@PreAuthorize("isAnonymous()")
public Account[] findAccounts();

@PreAuthorize("hasAuthority('ROLE_TELLER')")
public Account post(Account account, double amount);
}
```

Kotlin

```
interface BankService {
    @PreAuthorize("isAnonymous()")
    fun readAccount(id: Long): Account

    @PreAuthorize("isAnonymous()")
    fun findAccounts(): Array<Account>

    @PreAuthorize("hasAuthority('ROLE_TELLER')")
    fun post(account: Account, amount: Double): Account
}
```

11.5.3. GlobalMethodSecurityConfiguration

Sometimes you may need to perform operations that are more complicated than are possible with the `@EnableGlobalMethodSecurity` annotation allow. For these instances, you can extend the `GlobalMethodSecurityConfiguration` ensuring that the `@EnableGlobalMethodSecurity` annotation is present on your subclass. For example, if you wanted to provide a custom `MethodSecurityExpressionHandler`, you could use the following configuration:

Java

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {
    @Override
    protected MethodSecurityExpressionHandler createExpressionHandler() {
        // ... create and return custom MethodSecurityExpressionHandler ...
        return expressionHandler;
    }
}
```

Kotlin

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
open class MethodSecurityConfig : GlobalMethodSecurityConfiguration() {
    override fun createExpressionHandler(): MethodSecurityExpressionHandler {
        // ... create and return custom MethodSecurityExpressionHandler ...
        return expressionHandler
    }
}
```

For additional information about methods that can be overridden, refer to the `GlobalMethodSecurityConfiguration` Javadoc.

11.5.4. The <global-method-security> Element

This element is used to enable annotation-based security in your application (by setting the appropriate attributes on the element), and also to group together security pointcut declarations which will be applied across your entire application context. You should only declare one `<global-method-security>` element. The following declaration would enable support for Spring Security's `@Secured`:

```
<global-method-security secured-annotations="enabled" />
```

Adding an annotation to a method (on an class or interface) would then limit the access to that method accordingly. Spring Security's native annotation support defines a set of attributes for the method. These will be passed to the `AccessDecisionManager` for it to make the actual decision:

Java

```
public interface BankService {

@Secured("IS_AUTHENTICATED_ANONYMOUSLY")
public Account readAccount(Long id);

@Secured("IS_AUTHENTICATED_ANONYMOUSLY")
public Account[] findAccounts();

@Secured("ROLE_TELLER")
public Account post(Account account, double amount);
}
```

Kotlin

```
interface BankService {
    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    fun readAccount(id: Long): Account

    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    fun findAccounts(): Array<Account>

    @Secured("ROLE_TELLER")
    fun post(account: Account, amount: Double): Account
}
```

Support for JSR-250 annotations can be enabled using

```
<global-method-security jsr250-annotations="enabled" />
```

These are standards-based and allow simple role-based constraints to be applied but do not have the power Spring Security's native annotations. To use the new expression-based syntax, you would use

```
<global-method-security pre-post-annotations="enabled" />
```

and the equivalent Java code would be

Java

```
public interface BankService {

@PreAuthorize("isAnonymous()")
public Account readAccount(Long id);

@PreAuthorize("isAnonymous()")
public Account[] findAccounts();

@PreAuthorize("hasAuthority('ROLE_TELLER')")
public Account post(Account account, double amount);
}
```

Kotlin

```
interface BankService {
    @PreAuthorize("isAnonymous()")
    fun readAccount(id: Long): Account

    @PreAuthorize("isAnonymous()")
    fun findAccounts(): Array<Account>

    @PreAuthorize("hasAuthority('ROLE_TELLER')")
    fun post(account: Account, amount: Double): Account
}
```

Expression-based annotations are a good choice if you need to define simple rules that go beyond checking the role names against the user's list of authorities.

The annotated methods will only be secured for instances which are defined as Spring beans (in the same application context in which method-security is enabled). If you want to secure instances which are not created by Spring (using the `new` operator, for example) then you need to use AspectJ.

You can enable more than one type of annotation in the same application, but only one type should be used for any interface or class as the behaviour will not be well-defined otherwise. If two annotations are found which apply to a particular method, then only one of them will be applied.

11.5.5. Adding Security Pointcuts using protect-pointcut

The use of `protect-pointcut` is particularly powerful, as it allows you to apply security to many beans with only a simple declaration. Consider the following example:

```
<global-method-security>
<protect-pointcut expression="execution(* com.mycompany.*Service.*(..))"
    access="ROLE_USER"/>
</global-method-security>
```

This will protect all methods on beans declared in the application context whose classes are in the `com.mycompany` package and whose class names end in "Service". Only users with the `ROLE_USER` role will be able to invoke these methods. As with URL matching, the most specific matches must come first in the list of pointcuts, as the first matching expression will be used. Security annotations take precedence over pointcuts.

11.6. Domain Object Security (ACLs)

11.6.1. Overview

Complex applications often will find the need to define access permissions not simply at a web request or method invocation level. Instead, security decisions need to comprise both who (`Authentication`), where (`MethodInvocation`) and what (`SomeDomainObject`). In other words, authorization decisions also need to consider the actual domain object instance subject of a method invocation.

Imagine you're designing an application for a pet clinic. There will be two main groups of users of your Spring-based application: staff of the pet clinic, as well as the pet clinic's customers. The staff will have access to all of the data, whilst your customers will only be able to see their own customer records. To make it a little more interesting, your customers can allow other users to see their customer records, such as their "puppy preschool" mentor or president of their local "Pony Club". Using Spring Security as the foundation, you have several approaches that can be used:

- Write your business methods to enforce the security. You could consult a collection within the `Customer` domain object instance to determine which users have access. By using the `SecurityContextHolder.getContext().getAuthentication()`, you'll be able to access the `Authentication` object.
- Write an `AccessDecisionVoter` to enforce the security from the `GrantedAuthority[]` s stored in the `Authentication` object. This would mean your `AuthenticationManager` would need to populate the `Authentication` with custom `GrantedAuthority[]`s representing each of the `Customer` domain object instances the principal has access to.
- Write an `AccessDecisionVoter` to enforce the security and open the target `Customer` domain object directly. This would mean your voter needs access to a DAO that allows it to retrieve the `Customer` object. It would then access the `Customer` object's collection of approved users and make the appropriate decision.

Each one of these approaches is perfectly legitimate. However, the first couples your authorization checking to your business code. The main problems with this include the enhanced difficulty of unit testing and the fact it would be more difficult to reuse the `Customer` authorization logic elsewhere. Obtaining the `GrantedAuthority[]` s from the `Authentication` object is also fine, but will not scale to large numbers of `Customer` s. If a user might be able to access 5,000 `Customer` s (unlikely

in this case, but imagine if it were a popular vet for a large Pony Club!) the amount of memory consumed and time required to construct the `Authentication` object would be undesirable. The final method, opening the `Customer` directly from external code, is probably the best of the three. It achieves separation of concerns, and doesn't misuse memory or CPU cycles, but it is still inefficient in that both the `AccessDecisionVoter` and the eventual business method itself will perform a call to the DAO responsible for retrieving the `Customer` object. Two accesses per method invocation is clearly undesirable. In addition, with every approach listed you'll need to write your own access control list (ACL) persistence and business logic from scratch.

Fortunately, there is another alternative, which we'll talk about below.

11.6.2. Key Concepts

Spring Security's ACL services are shipped in the `spring-security-acl-xxx.jar`. You will need to add this JAR to your classpath to use Spring Security's domain object instance security capabilities.

Spring Security's domain object instance security capabilities centre on the concept of an access control list (ACL). Every domain object instance in your system has its own ACL, and the ACL records details of who can and can't work with that domain object. With this in mind, Spring Security delivers three main ACL-related capabilities to your application:

- A way of efficiently retrieving ACL entries for all of your domain objects (and modifying those ACLs)
- A way of ensuring a given principal is permitted to work with your objects, before methods are called
- A way of ensuring a given principal is permitted to work with your objects (or something they return), after methods are called

As indicated by the first bullet point, one of the main capabilities of the Spring Security ACL module is providing a high-performance way of retrieving ACLs. This ACL repository capability is extremely important, because every domain object instance in your system might have several access control entries, and each ACL might inherit from other ACLs in a tree-like structure (this is supported out-of-the-box by Spring Security, and is very commonly used). Spring Security's ACL capability has been carefully designed to provide high performance retrieval of ACLs, together with pluggable caching, deadlock-minimizing database updates, independence from ORM frameworks (we use JDBC directly), proper encapsulation, and transparent database updating.

Given databases are central to the operation of the ACL module, let's explore the four main tables used by default in the implementation. The tables are presented below in order of size in a typical Spring Security ACL deployment, with the table with the most rows listed last:

- ACL_SID allows us to uniquely identify any principal or authority in the system ("SID" stands for "security identity"). The only columns are the ID, a textual representation of the SID, and a flag to indicate whether the textual representation refers to a principal name or a `GrantedAuthority`. Thus, there is a single row for each unique principal or `GrantedAuthority`. When used in the context of receiving a permission, a SID is generally called a "recipient".
- ACL_CLASS allows us to uniquely identify any domain object class in the system. The only columns are the ID and the Java class name. Thus, there is a single row for each unique Class we

wish to store ACL permissions for.

- ACL_OBJECT_IDENTITY stores information for each unique domain object instance in the system. Columns include the ID, a foreign key to the ACL_CLASS table, a unique identifier so we know which ACL_CLASS instance we're providing information for, the parent, a foreign key to the ACL_SID table to represent the owner of the domain object instance, and whether we allow ACL entries to inherit from any parent ACL. We have a single row for every domain object instance we're storing ACL permissions for.
- Finally, ACL_ENTRY stores the individual permissions assigned to each recipient. Columns include a foreign key to the ACL_OBJECT_IDENTITY, the recipient (i.e. a foreign key to ACL_SID), whether we'll be auditing or not, and the integer bit mask that represents the actual permission being granted or denied. We have a single row for every recipient that receives a permission to work with a domain object.

As mentioned in the last paragraph, the ACL system uses integer bit masking. Don't worry, you need not be aware of the finer points of bit shifting to use the ACL system, but suffice to say that we have 32 bits we can switch on or off. Each of these bits represents a permission, and by default the permissions are read (bit 0), write (bit 1), create (bit 2), delete (bit 3) and administer (bit 4). It's easy to implement your own `Permission` instance if you wish to use other permissions, and the remainder of the ACL framework will operate without knowledge of your extensions.

It is important to understand that the number of domain objects in your system has absolutely no bearing on the fact we've chosen to use integer bit masking. Whilst you have 32 bits available for permissions, you could have billions of domain object instances (which will mean billions of rows in ACL_OBJECT_IDENTITY and quite probably ACL_ENTRY). We make this point because we've found sometimes people mistakenly believe they need a bit for each potential domain object, which is not the case.

Now that we've provided a basic overview of what the ACL system does, and what it looks like at a table structure, let's explore the key interfaces. The key interfaces are:

- `Acl`: Every domain object has one and only one `Acl` object, which internally holds the `AccessControlEntry` s as well as knows the owner of the `Acl`. An Acl does not refer directly to the domain object, but instead to an `ObjectIdentity`. The `Acl` is stored in the ACL_OBJECT_IDENTITY table.
- `AccessControlEntry`: An `Acl` holds multiple `AccessControlEntry` s, which are often abbreviated as ACEs in the framework. Each ACE refers to a specific tuple of `Permission`, `Sid` and `Acl`. An ACE can also be granting or non-granting and contain audit settings. The ACE is stored in the ACL_ENTRY table.
- `Permission`: A permission represents a particular immutable bit mask, and offers convenience functions for bit masking and outputting information. The basic permissions presented above (bits 0 through 4) are contained in the `BasePermission` class.
- `Sid`: The ACL module needs to refer to principals and `GrantedAuthority[]` s. A level of indirection is provided by the `Sid` interface, which is an abbreviation of "security identity". Common classes include `PrincipalSid` (to represent the principal inside an `Authentication` object) and `GrantedAuthoritySid`. The security identity information is stored in the ACL_SID table.
- `ObjectIdentity`: Each domain object is represented internally within the ACL module by an

`ObjectIdentity`. The default implementation is called `ObjectIdentityImpl`.

- `AclService`: Retrieves the `Acl` applicable for a given `ObjectIdentity`. In the included implementation (`JdbcAclService`), retrieval operations are delegated to a `LookupStrategy`. The `LookupStrategy` provides a highly optimized strategy for retrieving ACL information, using batched retrievals (`BasicLookupStrategy`) and supporting custom implementations that leverage materialized views, hierarchical queries and similar performance-centric, non-ANSI SQL capabilities.
- `MutableAclService`: Allows a modified `Acl` to be presented for persistence. It is not essential to use this interface if you do not wish.

Please note that our out-of-the-box AclService and related database classes all use ANSI SQL. This should therefore work with all major databases. At the time of writing, the system had been successfully tested using Hypersonic SQL, PostgreSQL, Microsoft SQL Server and Oracle.

Two samples ship with Spring Security that demonstrate the ACL module. The first is the Contacts Sample, and the other is the Document Management System (DMS) Sample. We suggest taking a look over these for examples.

11.6.3. Getting Started

To get starting using Spring Security's ACL capability, you will need to store your ACL information somewhere. This necessitates the instantiation of a `DataSource` using Spring. The `DataSource` is then injected into a `JdbcMutableAclService` and `BasicLookupStrategy` instance. The latter provides high-performance ACL retrieval capabilities, and the former provides mutator capabilities. Refer to one of the samples that ship with Spring Security for an example configuration. You'll also need to populate the database with the four ACL-specific tables listed in the last section (refer to the ACL samples for the appropriate SQL statements).

Once you've created the required schema and instantiated `JdbcMutableAclService`, you'll next need to ensure your domain model supports interoperability with the Spring Security ACL package. Hopefully `ObjectIdentityImpl` will prove sufficient, as it provides a large number of ways in which it can be used. Most people will have domain objects that contain a `public Serializable getId()` method. If the return type is long, or compatible with long (e.g. an int), you will find you need not give further consideration to `ObjectIdentity` issues. Many parts of the ACL module rely on long identifiers. If you're not using long (or an int, byte etc), there is a very good chance you'll need to reimplement a number of classes. We do not intend to support non-long identifiers in Spring Security's ACL module, as longs are already compatible with all database sequences, the most common identifier data type, and are of sufficient length to accommodate all common usage scenarios.

The following fragment of code shows how to create an `Acl`, or modify an existing `Acl`:

Java

```
// Prepare the information we'd like in our access control entry (ACE)
ObjectIdentity oi = new ObjectIdentityImpl(Foo.class, new Long(44));
Sid sid = new PrincipalSid("Samantha");
Permission p = BasePermission.ADMINISTRATION;

// Create or update the relevant ACL
MutableAcl acl = null;
try {
acl = (MutableAcl) aclService.readAclById(oi);
} catch (NotFoundException nfe) {
acl = aclService.createAcl(oi);
}

// Now grant some permissions via an access control entry (ACE)
acl.insertAce(acl.getEntries().length, p, sid, true);
aclService.updateAcl(acl);
```

Kotlin

```
val oi: ObjectIdentity = ObjectIdentityImpl(Foo::class.java, 44)
val sid: Sid = PrincipalSid("Samantha")
val p: Permission = BasePermission.ADMINISTRATION

// Create or update the relevant ACL
var acl: MutableAcl? = null
acl = try {
aclService.readAclById(oi) as MutableAcl
} catch (nfe: NotFoundException) {
aclService.createAcl(oi)
}

// Now grant some permissions via an access control entry (ACE)
acl!!.insertAce(acl.entries.size, p, sid, true)
aclService.updateAcl(acl)
```

In the example above, we're retrieving the ACL associated with the "Foo" domain object with identifier number 44. We're then adding an ACE so that a principal named "Samantha" can "administer" the object. The code fragment is relatively self-explanatory, except the insertAce method. The first argument to the insertAce method is determining at what position in the Acl the new entry will be inserted. In the example above, we're just putting the new ACE at the end of the existing ACEs. The final argument is a Boolean indicating whether the ACE is granting or denying. Most of the time it will be granting (true), but if it is denying (false), the permissions are effectively being blocked.

Spring Security does not provide any special integration to automatically create, update or delete ACLs as part of your DAO or repository operations. Instead, you will need to write code like shown

above for your individual domain objects. It's worth considering using AOP on your services layer to automatically integrate the ACL information with your services layer operations. We've found this quite an effective approach in the past.

Once you've used the above techniques to store some ACL information in the database, the next step is to actually use the ACL information as part of authorization decision logic. You have a number of choices here. You could write your own `AccessDecisionVoter` or `AfterInvocationProvider` that respectively fires before or after a method invocation. Such classes would use `AclService` to retrieve the relevant ACL and then call `Acl.isGranted(Permission[] permission, Sid[] sids, boolean administrativeMode)` to decide whether permission is granted or denied. Alternately, you could use our `AclEntryVoter`, `AclEntryAfterInvocationProvider` or `AclEntryAfterInvocationCollectionFilteringProvider` classes. All of these classes provide a declarative-based approach to evaluating ACL information at runtime, freeing you from needing to write any code. Please refer to the sample applications to learn how to use these classes.

Chapter 12. OAuth2

12.1. OAuth 2.0 Login

The OAuth 2.0 Login feature provides an application with the capability to have users log in to the application by using their existing account at an OAuth 2.0 Provider (e.g. GitHub) or OpenID Connect 1.0 Provider (such as Google). OAuth 2.0 Login implements the use cases: "Login with Google" or "Login with GitHub".

OAuth 2.0 Login is implemented by using the **Authorization Code Grant**, as specified in the OAuth 2.0 Authorization Framework and OpenID Connect Core 1.0.

12.1.1. Spring Boot 2.x Sample

Spring Boot 2.x brings full auto-configuration capabilities for OAuth 2.0 Login.

This section shows how to configure the **OAuth 2.0 Login sample** using *Google* as the *Authentication Provider* and covers the following topics:

- Initial setup
- Setting the redirect URI
- Configure application.yml
- Boot up the application

Initial setup

To use Google's OAuth 2.0 authentication system for login, you must set up a project in the Google API Console to obtain OAuth 2.0 credentials.

Google's OAuth 2.0 implementation for authentication conforms to the OpenID Connect 1.0 specification and is OpenID Certified.

Follow the instructions on the OpenID Connect page, starting in the section, "Setting up OAuth 2.0".

After completing the "Obtain OAuth 2.0 credentials" instructions, you should have a new OAuth Client with credentials consisting of a Client ID and a Client Secret.

Setting the redirect URI

The redirect URI is the path in the application that the end-user's user-agent is redirected back to after they have authenticated with Google and have granted access to the OAuth Client *(created in the previous step)* on the Consent page.

In the "Set a redirect URI" sub-section, ensure that the **Authorized redirect URIs** field is set to `http://localhost:8080/login/oauth2/code/google`.

The default redirect URI template is `{baseUrl}/login/oauth2/code/{registrationId}`. The ***registrationId*** is a unique identifier for the ClientRegistration.

If the OAuth Client is running behind a proxy server, it is recommended to check Proxy Server Configuration to ensure the application is correctly configured. Also, see the supported `URI` template variables for `redirect-uri`.

Configure application.yml

Now that you have a new OAuth Client with Google, you need to configure the application to use the OAuth Client for the *authentication flow*. To do so:

1. Go to `application.yml` and set the following configuration:

```
spring:
  security:
    oauth2:
      client:
        registration:   ①
          google:   ②
            client-id: google-client-id
            client-secret: google-client-secret
```

Example 104. OAuth Client properties

① `spring.security.oauth2.client.registration` is the base property prefix for OAuth Client properties.

② Following the base property prefix is the ID for the ClientRegistration, such as google.

2. Replace the values in the `client-id` and `client-secret` property with the OAuth 2.0 credentials you created earlier.

Boot up the application

Launch the Spring Boot 2.x sample and go to `http://localhost:8080`. You are then redirected to the default *auto-generated* login page, which displays a link for Google.

Click on the Google link, and you are then redirected to Google for authentication.

After authenticating with your Google account credentials, the next page presented to you is the Consent screen. The Consent screen asks you to either allow or deny access to the OAuth Client you created earlier. Click **Allow** to authorize the OAuth Client to access your email address and basic profile information.

At this point, the OAuth Client retrieves your email address and basic profile information from the UserInfo Endpoint and establishes an authenticated session.

12.1.2. Spring Boot 2.x Property Mappings

The following table outlines the mapping of the Spring Boot 2.x OAuth Client properties to the ClientRegistration properties.

Spring Boot 2.x	ClientRegistration
spring.security.oauth2.client.registration.[registrationId]	registrationId
spring.security.oauth2.client.registration.[registrationId].client-id	clientId
spring.security.oauth2.client.registration.[registrationId].client-secret	clientSecret
spring.security.oauth2.client.registration.[registrationId].client-authentication-method	clientAuthenticationMethod
spring.security.oauth2.client.registration.[registrationId].authorization-grant-type	authorizationGrantType
spring.security.oauth2.client.registration.[registrationId].redirect-uri	redirectUri
spring.security.oauth2.client.registration.[registrationId].scope	scopes
spring.security.oauth2.client.registration.[registrationId].client-name	clientName
spring.security.oauth2.client.provider.[providerId].authorization-uri	providerDetails.authorizationUri
spring.security.oauth2.client.provider.[providerId].token-uri	providerDetails.tokenUri
spring.security.oauth2.client.provider.[providerId].jwk-set-uri	providerDetails.jwkSetUri
spring.security.oauth2.client.provider.[providerId].issuer-uri	providerDetails.issuerUri
spring.security.oauth2.client.provider.[providerId].user-info-uri	providerDetails.userInfoEndpoint.uri
spring.security.oauth2.client.provider.[providerId].user-info-authentication-method	providerDetails.userInfoEndpoint.authenticationMethod
spring.security.oauth2.client.provider.[providerId].user-name-attribute	providerDetails.userInfoEndpoint.userNameAttributeName

A ClientRegistration can be initially configured using discovery of an OpenID Connect Provider's Configuration endpoint or an Authorization Server's Metadata endpoint, by specifying the spring.security.oauth2.client.provider.[providerId].issuer-uri property.

12.1.3. CommonOAuth2Provider

CommonOAuth2Provider pre-defines a set of default client properties for a number of well known providers: Google, GitHub, Facebook, and Okta.

For example, the `authorization-uri`, `token-uri`, and `user-info-uri` do not change often for a Provider. Therefore, it makes sense to provide default values in order to reduce the required configuration.

As demonstrated previously, when we configured a Google client, only the `client-id` and `client-secret` properties are required.

The following listing shows an example:

```
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id: google-client-id
            client-secret: google-client-secret
```

The auto-defaulting of client properties works seamlessly here because the `registrationId` (`google`) matches the `GOOGLE enum` (case-insensitive) in `CommonOAuth2Provider`.

For cases where you may want to specify a different `registrationId`, such as `google-login`, you can still leverage auto-defaulting of client properties by configuring the `provider` property.

The following listing shows an example:

```
spring:
  security:
    oauth2:
      client:
        registration:
          google-login: ①
            provider: google    ②
            client-id: google-client-id
            client-secret: google-client-secret
```

① The `registrationId` is set to `google-login`.

② The `provider` property is set to `google`, which will leverage the auto-defaulting of client properties set in `CommonOAuth2Provider.GOOGLE.getBuilder()`.

12.1.4. Configuring Custom Provider Properties

There are some OAuth 2.0 Providers that support multi-tenancy, which results in different protocol endpoints for each tenant (or sub-domain).

For example, an OAuth Client registered with Okta is assigned to a specific sub-domain and have their own protocol endpoints.

For these cases, Spring Boot 2.x provides the following base property for configuring custom provider properties: `spring.security.oauth2.client.provider.[providerId]`.

The following listing shows an example:

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: okta-client-id
            client-secret: okta-client-secret
        provider:
          okta: ①
            authorization-uri: https://your-
subdomain.oktapreview.com/oauth2/v1/authorize
            token-uri: https://your-subdomain.oktapreview.com/oauth2/v1/token
            user-info-uri: https://your-subdomain.oktapreview.com/oauth2/v1/userinfo
            user-name-attribute: sub
            jwk-set-uri: https://your-subdomain.oktapreview.com/oauth2/v1/keys
```

① The base property (`spring.security.oauth2.client.provider.okta`) allows for custom configuration of protocol endpoint locations.

12.1.5. Overriding Spring Boot 2.x Auto-configuration

The Spring Boot 2.x auto-configuration class for OAuth Client support is `OAuth2ClientAutoConfiguration`.

It performs the following tasks:

- Registers a `ClientRegistrationRepository` `@Bean` composed of `ClientRegistration`(s) from the configured OAuth Client properties.
- Provides a `WebSecurityConfigurerAdapter` `@Configuration` and enables OAuth 2.0 Login through `httpSecurity.oauth2Login()`.

If you need to override the auto-configuration based on your specific requirements, you may do so in the following ways:

- Register a ClientRegistrationRepository @Bean
- Provide a WebSecurityConfigurerAdapter
- Completely Override the Auto-configuration

Register a ClientRegistrationRepository @Bean

The following example shows how to register a `ClientRegistrationRepository` `@Bean`:

Java

```
@Configuration
public class OAuth2LoginConfig {

    @Bean
    public ClientRegistrationRepository clientRegistrationRepository() {
        return new
InMemoryClientRegistrationRepository(this.googleClientRegistration());
    }

    private ClientRegistration googleClientRegistration() {
        return ClientRegistration.withRegistrationId("google")
            .clientId("google-client-id")
            .clientSecret("google-client-secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .redirectUri("{baseUrl}/login/oauth2/code/{registrationId}")
            .scope("openid", "profile", "email", "address", "phone")
            .authorizationUri("https://accounts.google.com/o/oauth2/v2/auth")
            .tokenUri("https://www.googleapis.com/oauth2/v4/token")
            .userInfoUri("https://www.googleapis.com/oauth2/v3/userinfo")
            .userNameAttributeName(IdTokenClaimNames.SUB)
            .jwkSetUri("https://www.googleapis.com/oauth2/v3/certs")
            .clientName("Google")
            .build();
    }
}
```

Kotlin

```
@Configuration
class OAuth2LoginConfig {
    @Bean
    fun clientRegistrationRepository(): ClientRegistrationRepository {
        return InMemoryClientRegistrationRepository(googleClientRegistration())
    }

    private fun googleClientRegistration(): ClientRegistration {
        return ClientRegistration.withRegistrationId("google")
                .clientId("google-client-id")
                .clientSecret("google-client-secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
                .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
                .redirectUri("{baseUrl}/login/oauth2/code/{registrationId}")
                .scope("openid", "profile", "email", "address", "phone")
                .authorizationUri("https://accounts.google.com/o/oauth2/v2/auth")
                .tokenUri("https://www.googleapis.com/oauth2/v4/token")
                .userInfoUri("https://www.googleapis.com/oauth2/v3/userinfo")
                .userNameAttributeName(IdTokenClaimNames.SUB)
                .jwkSetUri("https://www.googleapis.com/oauth2/v3/certs")
                .clientName("Google")
                .build()
    }
}
```

Provide a WebSecurityConfigurerAdapter

The following example shows how to provide a `WebSecurityConfigurerAdapter` with `@EnableWebSecurity` and enable OAuth 2.0 login through `httpSecurity.oauth2Login()`:

Example 105. OAuth2 Login Configuration

Java

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .oauth2Login(withDefaults());
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2LoginSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            authorizeRequests {
                authorize(anyRequest, authenticated)
            }
            oauth2Login { }
        }
    }
}
```

Completely Override the Auto-configuration

The following example shows how to completely override the auto-configuration by registering a `ClientRegistrationRepository` `@Bean` and providing a `WebSecurityConfigurerAdapter`.

Example 106. Overriding the auto-configuration

Java

```
@Configuration
public class OAuth2LoginConfig {

    @EnableWebSecurity
    public static class OAuth2LoginSecurityConfig extends
WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests(authorize -> authorize
                    .anyRequest().authenticated()
                )
                .oauth2Login(withDefaults());
        }
    }

    @Bean
    public ClientRegistrationRepository clientRegistrationRepository() {
        return new
InMemoryClientRegistrationRepository(this.googleClientRegistration());
    }

    private ClientRegistration googleClientRegistration() {
        return ClientRegistration.withRegistrationId("google")
            .clientId("google-client-id")
            .clientSecret("google-client-secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .redirectUri("{baseUrl}/login/oauth2/code/{registrationId}")
            .scope("openid", "profile", "email", "address", "phone")
            .authorizationUri("https://accounts.google.com/o/oauth2/v2/auth")
            .tokenUri("https://www.googleapis.com/oauth2/v4/token")
            .userInfoUri("https://www.googleapis.com/oauth2/v3/userinfo")
            .userNameAttributeName(IdTokenClaimNames.SUB)
            .jwkSetUri("https://www.googleapis.com/oauth2/v3/certs")
            .clientName("Google")
            .build();
    }
}
```

Kotlin

```
@Configuration
class OAuth2LoginConfig {

    @EnableWebSecurity
    class OAuth2LoginSecurityConfig: WebSecurityConfigurerAdapter() {

        override fun configure(http: HttpSecurity) {
            http {
                authorizeRequests {
                    authorize(anyRequest, authenticated)
                }
                oauth2Login { }
            }
        }
    }

    @Bean
    fun clientRegistrationRepository(): ClientRegistrationRepository {
        return InMemoryClientRegistrationRepository(googleClientRegistration())
    }

    private fun googleClientRegistration(): ClientRegistration {
        return ClientRegistration.withRegistrationId("google")
                .clientId("google-client-id")
                .clientSecret("google-client-secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
                .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
                .redirectUri("{baseUrl}/login/oauth2/code/{registrationId}")
                .scope("openid", "profile", "email", "address", "phone")
                .authorizationUri("https://accounts.google.com/o/oauth2/v2/auth")
                .tokenUri("https://www.googleapis.com/oauth2/v4/token")
                .userInfoUri("https://www.googleapis.com/oauth2/v3/userinfo")
                .userNameAttributeName(IdTokenClaimNames.SUB)
                .jwkSetUri("https://www.googleapis.com/oauth2/v3/certs")
                .clientName("Google")
                .build()
    }
}
```

12.1.6. Java Configuration without Spring Boot 2.x

If you are not able to use Spring Boot 2.x and would like to configure one of the pre-defined providers in `CommonOAuth2Provider` (for example, Google), apply the following configuration:

Example 107. OAuth2 Login Configuration

Java

```
@Configuration
public class OAuth2LoginConfig {

    @EnableWebSecurity
    public static class OAuth2LoginSecurityConfig extends
WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests(authorize -> authorize
                    .anyRequest().authenticated()
                )
                .oauth2Login(withDefaults());
        }
    }

    @Bean
    public ClientRegistrationRepository clientRegistrationRepository() {
        return new
InMemoryClientRegistrationRepository(this.googleClientRegistration());
    }

    @Bean
    public OAuth2AuthorizedClientService authorizedClientService(
            ClientRegistrationRepository clientRegistrationRepository) {
        return new
InMemoryOAuth2AuthorizedClientService(clientRegistrationRepository);
    }

    @Bean
    public OAuth2AuthorizedClientRepository authorizedClientRepository(
            OAuth2AuthorizedClientService authorizedClientService) {
        return new
AuthenticatedPrincipalOAuth2AuthorizedClientRepository(authorizedClientService);
    }

    private ClientRegistration googleClientRegistration() {
        return CommonOAuth2Provider.GOOGLE.getBuilder("google")
            .clientId("google-client-id")
            .clientSecret("google-client-secret")
            .build();
    }
}
```

Kotlin

```
@Configuration
open class OAuth2LoginConfig {
    @EnableWebSecurity
    open class OAuth2LoginSecurityConfig : WebSecurityConfigurerAdapter() {
        override fun configure(http: HttpSecurity) {
            http {
                authorizeRequests {
                    authorize(anyRequest, authenticated)
                }
                oauth2Login { }
            }
        }
    }

    @Bean
    open fun clientRegistrationRepository(): ClientRegistrationRepository {
        return InMemoryClientRegistrationRepository(googleClientRegistration())
    }

    @Bean
    open fun authorizedClientService(
        clientRegistrationRepository: ClientRegistrationRepository?
    ): OAuth2AuthorizedClientService {
        return InMemoryOAuth2AuthorizedClientService(clientRegistrationRepository)
    }

    @Bean
    open fun authorizedClientRepository(
        authorizedClientService: OAuth2AuthorizedClientService?
    ): OAuth2AuthorizedClientRepository {
        return
AuthenticatedPrincipalOAuth2AuthorizedClientRepository(authorizedClientService)
    }

    private fun googleClientRegistration(): ClientRegistration {
        return CommonOAuth2Provider.GOOGLE.getBuilder("google")
            .clientId("google-client-id")
            .clientSecret("google-client-secret")
            .build()
    }
}
```

Xml

```
<http auto-config="true">
    <intercept-url pattern="/**" access="authenticated"/>
    <oauth2-login authorized-client-repository-ref="authorizedClientRepository"/>
</http>

<client-registrations>
    <client-registration registration-id="google"
                         client-id="google-client-id"
                         client-secret="google-client-secret"
                         provider-id="google"/>
</client-registrations>

<b:bean id="authorizedClientService"

class="org.springframework.security.oauth2.client.InMemoryOAuth2AuthorizedClientSe
rvice"
        autowire="constructor"/>

<b:bean id="authorizedClientRepository"

class="org.springframework.security.oauth2.client.web.AuthenticatedPrincipalOAuth2
AuthorizedClientRepository">
    <b:constructor-arg ref="authorizedClientService"/>
</b:bean>
```

12.1.7. Advanced Configuration

`HttpSecurity.oauth2Login()` provides a number of configuration options for customizing OAuth 2.0 Login. The main configuration options are grouped into their protocol endpoint counterparts.

For example, `oauth2Login().authorizationEndpoint()` allows configuring the *Authorization Endpoint*, whereas `oauth2Login().tokenEndpoint()` allows configuring the *Token Endpoint*.

The following code shows an example:

Example 108. Advanced OAuth2 Login Configuration

Java

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login(oauth2 -> oauth2
                .authorizationEndpoint(authorization -> authorization
                        ...
                )
                .redirectionEndpoint(redirection -> redirection
                        ...
                )
                .tokenEndpoint(token -> token
                        ...
                )
                .userInfoEndpoint(userInfo -> userInfo
                        ...
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2LoginSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            oauth2Login {
                authorizationEndpoint {
                    ...
                }
                redirectionEndpoint {
                    ...
                }
                tokenEndpoint {
                    ...
                }
                userInfoEndpoint {
                    ...
                }
            }
        }
    }
}
```

The main goal of the `oauth2Login()` DSL was to closely align with the naming, as defined in the specifications.

The OAuth 2.0 Authorization Framework defines the Protocol Endpoints as follows:

The authorization process utilizes two authorization server endpoints (HTTP resources):

- Authorization Endpoint: Used by the client to obtain authorization from the resource owner via user-agent redirection.
- Token Endpoint: Used by the client to exchange an authorization grant for an access token, typically with client authentication.

As well as one client endpoint:

- Redirection Endpoint: Used by the authorization server to return responses containing authorization credentials to the client via the resource owner user-agent.

The OpenID Connect Core 1.0 specification defines the UserInfo Endpoint as follows:

The UserInfo Endpoint is an OAuth 2.0 Protected Resource that returns claims about the authenticated end-user. To obtain the requested claims about the end-user, the client makes a request to the UserInfo Endpoint by using an access token obtained through OpenID Connect Authentication. These claims are normally represented by a JSON object that contains a collection of name-value pairs for the claims.

The following code shows the complete configuration options available for the `oauth2Login()` DSL:

Example 109. OAuth2 Login Configuration Options

Java

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login(oauth2 -> oauth2
                .clientRegistrationRepository(this.clientRegistrationRepository())
                .authorizedClientRepository(this.authorizedClientRepository())
                .authorizedClientService(this.authorizedClientService())
                .loginPage("/login")
                .authorizationEndpoint(authorization -> authorization
                    .baseUri(this.authorizationRequestBaseUri())

.authorizationRequestRepository(this.authorizationRequestRepository())

.authorizationRequestResolver(this.authorizationRequestResolver())
                )
                .redirectionEndpoint(redirection -> redirection
                    .baseUri(this.authorizationResponseBaseUri())
                )
                .tokenEndpoint(token -> token
                    .accessTokenResponseClient(this.accessTokenResponseClient())
                )
                .userInfoEndpoint(userInfo -> userInfo
                    .userAuthoritiesMapper(this.userAuthoritiesMapper())
                    .userService(this.oauth2UserService())
                    .oidcUserService(this.oidcUserService())
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2LoginSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            oauth2Login {
                clientRegistrationRepository = clientRegistrationRepository()
                authorizedClientRepository = authorizedClientRepository()
                authorizedClientService = authorizedClientService()
                loginPage = "/login"
                authorizationEndpoint {
                    baseUri = authorizationRequestBaseUri()
                    authorizationRequestRepository =
authorizationRequestRepository()
                    authorizationRequestResolver = authorizationRequestResolver()
                }
                redirectionEndpoint {
                    baseUri = authorizationResponseBaseUri()
                }
                tokenEndpoint {
                    accessTokenResponseClient = accessTokenResponseClient()
                }
                userInfoEndpoint {
                    userAuthoritiesMapper = userAuthoritiesMapper()
                    userService = oauth2UserService()
                    oidcUserService = oidcUserService()
                }
            }
        }
    }
}
```

In addition to the `oauth2Login()` DSL, XML configuration is also supported.

The following code shows the complete configuration options available in the security namespace:

Example 110. OAuth2 Login XML Configuration Options

```
<http>
    <oauth2-login client-registration-repository-
ref="clientRegistrationRepository"
                  authorized-client-repository-ref="authorizedClientRepository"
                  authorized-client-service-ref="authorizedClientService"
                  authorization-request-repository-
ref="authorizationRequestRepository"
                  authorization-request-resolver-
ref="authorizationRequestResolver"
                  access-token-response-client-ref="accessTokenResponseClient"
                  user-authorities-mapper-ref="userAuthoritiesMapper"
                  user-service-ref="oauth2UserService"
                  oidc-user-service-ref="oidcUserService"
                  login-processing-url="/login/oauth2/code/*"
                  login-page="/login"
                  authentication-success-handler-
ref="authenticationSuccessHandler"
                  authentication-failure-handler-
ref="authenticationFailureHandler"
                  jwt-decoder-factory-ref="jwtDecoderFactory"/>
</http>
```

The following sections go into more detail on each of the configuration options available:

- OAuth 2.0 Login Page
- Redirection Endpoint
- UserInfo Endpoint

OAuth 2.0 Login Page

By default, the OAuth 2.0 Login Page is auto-generated by the `DefaultLoginPageGeneratingFilter`. The default login page shows each configured OAuth Client with its `ClientRegistration.clientName` as a link, which is capable of initiating the Authorization Request (or OAuth 2.0 Login).

In order for `DefaultLoginPageGeneratingFilter` to show links for configured OAuth Clients, the registered `ClientRegistrationRepository` needs to also implement `Iterable<ClientRegistration>`. See `InMemoryClientRegistrationRepository` for reference.

The link's destination for each OAuth Client defaults to the following:

```
OAuth2AuthorizationRequestRedirectFilter.DEFAULT_AUTHORIZATION_REQUEST_BASE_URI + "/{registrationId}"
```

The following line shows an example:

```
<a href="/oauth2/authorization/google">Google</a>
```

To override the default login page, configure `oauth2Login().loginPage()` and (optionally) `oauth2Login().authorizationEndpoint().baseUri()`.

The following listing shows an example:

Example 111. OAuth2 Login Page Configuration

Java

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login(oauth2 -> oauth2
                .loginPage("/login/oauth2")
                ...
                .authorizationEndpoint(authorization -> authorization
                    .baseUri("/login/oauth2/authorization")
                    ...
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2LoginSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            oauth2Login {
                loginPage = "/login/oauth2"
                authorizationEndpoint {
                    baseUri = "/login/oauth2/authorization"
                }
            }
        }
    }
}
```

Xml

```
<http>
    <oauth2-login login-page="/login/oauth2"
                  ...
    />
</http>
```

You need to provide a `@Controller` with a `@RequestMapping("/login/oauth2")` that is capable of rendering the custom login page.

As noted earlier, configuring `oauth2Login().authorizationEndpoint().baseUri()` is optional. However, if you choose to customize it, ensure the link to each OAuth Client matches the `authorizationEndpoint().baseUri()`.

The following line shows an example:

```
<a href="/login/oauth2/authorization/google">Google</a>
```

Redirection Endpoint

The Redirection Endpoint is used by the Authorization Server for returning the Authorization Response (which contains the authorization credentials) to the client via the Resource Owner user-agent.

OAuth 2.0 Login leverages the Authorization Code Grant. Therefore, the authorization credential is the authorization code.

The default Authorization Response `baseUri` (redirection endpoint) is **`/login/oauth2/code/*`**, which is defined in `OAuth2LoginAuthenticationFilter.DEFAULT_FILTER_PROCESSES_URI`.

If you would like to customize the Authorization Response `baseUri`, configure it as shown in the following example:

Example 112. Redirection Endpoint Configuration

Java

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login(oauth2 -> oauth2
                .redirectionEndpoint(redirection -> redirection
                    .baseUri("/login/oauth2/callback/*")
                    ...
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2LoginSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            oauth2Login {
                redirectionEndpoint {
                    baseUri = "/login/oauth2/callback/*"
                }
            }
        }
    }
}
```

Xml

```
<http>
    <oauth2-login login-processing-url="/login/oauth2/callback/*"
                  ...
    />
</http>
```

You also need to ensure the `ClientRegistration.redirectUri` matches the custom Authorization Response `baseUri`.

The following listing shows an example:

Java

```
return CommonOAuth2Provider.GOOGLE.getBuilder("google")
    .clientId("google-client-id")
    .clientSecret("google-client-secret")
    .redirectUri("{baseUrl}/login/oauth2/callback/{registrationId}")
    .build();
```

Kotlin

```
return CommonOAuth2Provider.GOOGLE.getBuilder("google")
    .clientId("google-client-id")
    .clientSecret("google-client-secret")
    .redirectUri("{baseUrl}/login/oauth2/callback/{registrationId}")
    .build()
```

UserInfo Endpoint

The UserInfo Endpoint includes a number of configuration options, as described in the following sub-sections:

- Mapping User Authorities
- OAuth 2.0 UserService
- OpenID Connect 1.0 UserService

Mapping User Authorities

After the user successfully authenticates with the OAuth 2.0 Provider, the `OAuth2User.getAuthorities()` (or `OidcUser.getAuthorities()`) may be mapped to a new set of `GrantedAuthority` instances, which will be supplied to `OAuth2AuthenticationToken` when completing the authentication.

`OAuth2AuthenticationToken.getAuthorities()` is used for authorizing requests, such as in `hasRole('USER')` or `hasRole('ADMIN')`.

There are a couple of options to choose from when mapping user authorities:

- Using a GrantedAuthoritiesMapper
- Delegation-based strategy with OAuth2UserService

Using a GrantedAuthoritiesMapper

Provide an implementation of `GrantedAuthoritiesMapper` and configure it as shown in the following

example:

Example 113. Granted Authorities Mapper Configuration

Java

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login(oauth2 -> oauth2
                .userInfoEndpoint(userInfo -> userInfo
                    .userAuthoritiesMapper(this.userAuthoritiesMapper())
                    ...
                )
            );
    }

    private GrantedAuthoritiesMapper userAuthoritiesMapper() {
        return (authorities) -> {
            Set<GrantedAuthority> mappedAuthorities = new HashSet<>();

            authorities.forEach(authority -> {
                if (OidcUserAuthority.class.isInstance(authority)) {
                    OidcUserAuthority oidcUserAuthority =
(OidcUserAuthority)authority;

                    OidcIdToken idToken = oidcUserAuthority.getIdToken();
                    OidcUserInfo userInfo = oidcUserAuthority.getUserInfo();

                    // Map the claims found in idToken and/or userInfo
                    // to one or more GrantedAuthority's and add it to
mappedAuthorities

                } else if (OAuth2UserAuthority.class.isInstance(authority)) {
                    OAuth2UserAuthority oauth2UserAuthority =
(OAuth2UserAuthority)authority;

                    Map<String, Object> userAttributes =
oauth2UserAuthority.getAttributes();

                    // Map the attributes found in userAttributes
                    // to one or more GrantedAuthority's and add it to
mappedAuthorities

                }
            });

            return mappedAuthorities;
        };
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2LoginSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            oauth2Login {
                userInfoEndpoint {
                    userAuthoritiesMapper = userAuthoritiesMapper()
                }
            }
        }
    }

    private fun userAuthoritiesMapper(): GrantedAuthoritiesMapper =
GrantedAuthoritiesMapper { authorities: Collection<GrantedAuthority> ->
        val mappedAuthorities = emptySet<GrantedAuthority>()

        authorities.forEach { authority ->
            if (authority is OidcUserAuthority) {
                val idToken = authority.idToken
                val userInfo = authority.userInfo
                // Map the claims found in idToken and/or userInfo
                // to one or more GrantedAuthority's and add it to
mappedAuthorities
            } else if (authority is OAuth2UserAuthority) {
                val userAttributes = authority.attributes
                // Map the attributes found in userAttributes
                // to one or more GrantedAuthority's and add it to
mappedAuthorities
            }
        }

        mappedAuthorities
    }
}
```

Xml

```
<http>
    <oauth2-login user-authorities-mapper-ref="userAuthoritiesMapper"
                  ...
    />
</http>
```

Alternatively, you may register a `GrantedAuthoritiesMapper` `@Bean` to have it automatically applied to the configuration, as shown in the following example:

Example 114. Granted Authorities Mapper Bean Configuration

Java

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login(withDefaults());
    }

    @Bean
    public GrantedAuthoritiesMapper userAuthoritiesMapper() {
        ...
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2LoginSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            oauth2Login { }
        }
    }

    @Bean
    fun userAuthoritiesMapper(): GrantedAuthoritiesMapper {
        ...
    }
}
```

Delegation-based strategy with OAuth2UserService

This strategy is advanced compared to using a `GrantedAuthoritiesMapper`, however, it's also more flexible as it gives you access to the `OAuth2UserRequest` and `OAuth2User` (when using an OAuth 2.0 UserService) or `OidcUserRequest` and `OidcUser` (when using an OpenID Connect 1.0 UserService).

The `OAuth2UserRequest` (and `OidcUserRequest`) provides you access to the associated `OAuth2AccessToken`, which is very useful in the cases where the *delegator* needs to fetch authority information from a protected resource before it can map the custom authorities for the user.

The following example shows how to implement and configure a delegation-based strategy using an OpenID Connect 1.0 UserService:

Example 115. OAuth2UserService Configuration

Java

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login(oauth2 -> oauth2
                .userInfoEndpoint(userInfo -> userInfo
                    .oidcUserService(this.oidcUserService())
                    ...
                )
            );
    }

    private OAuth2UserService<OidcUserRequest, OidcUser> oidcUserService() {
        final OidcUserService delegate = new OidcUserService();

        return (userRequest) -> {
            // Delegate to the default implementation for loading a user
            OidcUser oidcUser = delegate.loadUser(userRequest);

            OAuth2AccessToken accessToken = userRequest.getAccessToken();
            Set<GrantedAuthority> mappedAuthorities = new HashSet<>();

            // TODO
            // 1) Fetch the authority information from the protected resource
using accessToken
            // 2) Map the authority information to one or more GrantedAuthority's
and add it to mappedAuthorities

            // 3) Create a copy of oidcUser but use the mappedAuthorities instead
            oidcUser = new DefaultOidcUser(mappedAuthorities,
oidcUser.getIdToken(), oidcUser.getUserInfo());

            return oidcUser;
        };
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2LoginSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            oauth2Login {
                userInfoEndpoint {
                    oidcUserService = oidcUserService()
                }
            }
        }
    }

    @Bean
    fun oidcUserService(): OAuth2UserService<OidcUserRequest, OidcUser> {
        val delegate = OidcUserService()

        return OAuth2UserService { userRequest ->
            // Delegate to the default implementation for loading a user
            var oidcUser = delegate.loadUser(userRequest)

            val accessToken = userRequest.accessToken
            val mappedAuthorities = HashSet<GrantedAuthority>()

            // TODO
            // 1) Fetch the authority information from the protected resource
using accessToken
            // 2) Map the authority information to one or more GrantedAuthority's
and add it to mappedAuthorities
            // 3) Create a copy of oidcUser but use the mappedAuthorities instead
            oidcUser = DefaultOidcUser(mappedAuthorities, oidcUser.idToken,
oidcUser.userInfo)

            oidcUser
        }
    }
}
```

Xml

```
<http>
    <oauth2-login oidc-user-service-ref="oidcUserService"
                  ...
    />
</http>
```

OAuth 2.0 UserService

`DefaultOAuth2UserService` is an implementation of an `OAuth2UserService` that supports standard OAuth 2.0 Provider's.

`OAuth2UserService` obtains the user attributes of the end-user (the resource owner) from the UserInfo Endpoint (by using the access token granted to the client during the authorization flow) and returns an `AuthenticatedPrincipal` in the form of an `OAuth2User`.

`DefaultOAuth2UserService` uses a `RestOperations` when requesting the user attributes at the UserInfo Endpoint.

If you need to customize the pre-processing of the UserInfo Request, you can provide `DefaultOAuth2UserService.setRequestEntityConverter()` with a custom `Converter<OAuth2UserRequest, RequestEntity<?>>`. The default implementation `OAuth2UserRequestEntityConverter` builds a `RequestEntity` representation of a UserInfo Request that sets the `OAuth2AccessToken` in the `Authorization` header by default.

On the other end, if you need to customize the post-handling of the UserInfo Response, you will need to provide `DefaultOAuth2UserService.setRestOperations()` with a custom configured `RestOperations`. The default `RestOperations` is configured as follows:

```
RestTemplate restTemplate = new RestTemplate();
restTemplate.setErrorHandler(new OAuth2ErrorResponseErrorHandler());
```

`OAuth2ErrorResponseErrorHandler` is a `ResponseErrorHandler` that can handle an OAuth 2.0 Error (400 Bad Request). It uses an `OAuth2ErrorHttpMessageConverter` for converting the OAuth 2.0 Error parameters to an `OAuth2Error`.

Whether you customize `DefaultOAuth2UserService` or provide your own implementation of `OAuth2UserService`, you'll need to configure it as shown in the following example:

Java

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login(oauth2 -> oauth2
                .userInfoEndpoint(userInfo -> userInfo
                    .userService(this.oauth2UserService())
                    ...
                )
            );
    }

    private OAuth2UserService<OAuth2UserRequest, OAuth2User> oauth2UserService() {
        ...
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2LoginSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            oauth2Login {
                userInfoEndpoint {
                    userService = oauth2UserService()
                    // ...
                }
            }
        }
    }

    private fun oauth2UserService(): OAuth2UserService<OAuth2UserRequest,
OAuth2User> {
        // ...
    }
}
```

OpenID Connect 1.0 UserService

`OidcUserService` is an implementation of an `OAuth2UserService` that supports OpenID Connect 1.0 Provider's.

The `OidcUserService` leverages the `DefaultOAuth2UserService` when requesting the user attributes at the UserInfo Endpoint.

If you need to customize the pre-processing of the UserInfo Request and/or the post-handling of the UserInfo Response, you will need to provide `OidcUserService.setOauth2UserService()` with a custom configured `DefaultOAuth2UserService`.

Whether you customize `OidcUserService` or provide your own implementation of `OAuth2UserService` for OpenID Connect 1.0 Provider's, you'll need to configure it as shown in the following example:

Java

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login(oauth2 -> oauth2
                .userInfoEndpoint(userInfo -> userInfo
                    .oidcUserService(this.oidcUserService())
                    ...
                )
            );
    }

    private OAuth2UserService<OidcUserRequest, OidcUser> oidcUserService() {
        ...
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2LoginSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            oauth2Login {
                userInfoEndpoint {
                    oidcUserService = oidcUserService()
                    // ...
                }
            }
        }
    }

    private fun oidcUserService(): OAuth2UserService<OidcUserRequest, OidcUser> {
        // ...
    }
}
```

ID Token Signature Verification

OpenID Connect 1.0 Authentication introduces the ID Token, which is a security token that contains Claims about the Authentication of an End-User by an Authorization Server when used by a Client.

The ID Token is represented as a JSON Web Token (JWT) and MUST be signed using JSON Web

Signature (JWS).

The `OidcIdTokenDecoderFactory` provides a `JwtDecoder` used for `OidcIdToken` signature verification. The default algorithm is `RS256` but may be different when assigned during client registration. For these cases, a resolver may be configured to return the expected JWS algorithm assigned for a specific client.

The JWS algorithm resolver is a `Function` that accepts a `ClientRegistration` and returns the expected `JwsAlgorithm` for the client, eg. `SignatureAlgorithm.RS256` or `MacAlgorithm.HS256`

The following code shows how to configure the `OidcIdTokenDecoderFactory @Bean` to default to `MacAlgorithm.HS256` for all `ClientRegistration`:

Java

```
@Bean
public JwtDecoderFactory<ClientRegistration> idTokenDecoderFactory() {
    OidcIdTokenDecoderFactory idTokenDecoderFactory = new
OidcIdTokenDecoderFactory();
    idTokenDecoderFactory.setJwsAlgorithmResolver(clientRegistration ->
MacAlgorithm.HS256);
    return idTokenDecoderFactory;
}
```

Kotlin

```
@Bean
fun idTokenDecoderFactory(): JwtDecoderFactory<ClientRegistration?> {
    val idTokenDecoderFactory = OidcIdTokenDecoderFactory()
    idTokenDecoderFactory.setJwsAlgorithmResolver { MacAlgorithm.HS256 }
    return idTokenDecoderFactory
}
```

For MAC based algorithms such as `HS256`, `HS384` or `HS512`, the `client-secret` corresponding to the `client-id` is used as the symmetric key for signature verification.

If more than one `ClientRegistration` is configured for OpenID Connect 1.0 Authentication, the JWS algorithm resolver may evaluate the provided `ClientRegistration` to determine which algorithm to return.

OpenID Connect 1.0 Logout

OpenID Connect Session Management 1.0 allows the ability to log out the End-User at the Provider using the Client. One of the strategies available is RP-Initiated Logout.

If the OpenID Provider supports both Session Management and Discovery, the client may obtain the

`end_session_endpoint` `URL` from the OpenID Provider's Discovery Metadata. This can be achieved by configuring the `ClientRegistration` with the `issuer-uri`, as in the following example:

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: okta-client-id
            client-secret: okta-client-secret
            ...
        provider:
          okta:
            issuer-uri: https://dev-1234.oktapreview.com
```

...and the `OidcClientInitiatedLogoutSuccessHandler`, which implements RP-Initiated Logout, may be configured as follows:

Java

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private ClientRegistrationRepository clientRegistrationRepository;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .oauth2Login(withDefaults())
            .logout(logout -> logout
                .logoutSuccessHandler(oidcLogoutSuccessHandler())
            );
    }

    private LogoutSuccessHandler oidcLogoutSuccessHandler() {
        OidcClientInitiatedLogoutSuccessHandler oidcLogoutSuccessHandler =
                new
OidcClientInitiatedLogoutSuccessHandler(this.clientRegistrationRepository);

        // Sets the location that the End-User's User Agent will be redirected to
        // after the logout has been performed at the Provider
        oidcLogoutSuccessHandler.setPostLogoutRedirectUri("{baseUrl}");

        return oidcLogoutSuccessHandler;
    }
}

NOTE: `OidcClientInitiatedLogoutSuccessHandler` supports the `{baseUrl}`
placeholder.
If used, the application's base URL, like `https://app.example.org`, will replace
it at request time.
```

Kotlin

```
@EnableWebSecurity
class OAuth2LoginSecurityConfig : WebSecurityConfigurerAdapter() {
    @Autowired
    private lateinit var clientRegistrationRepository:
ClientRegistrationRepository

    override fun configure(http: HttpSecurity) {
        http {
            authorizeRequests {
                authorize(anyRequest, authenticated)
            }
            oauth2Login { }
            logout {
                logoutSuccessHandler = oidcLogoutSuccessHandler()
            }
        }
    }

    private fun oidcLogoutSuccessHandler(): LogoutSuccessHandler {
        val oidcLogoutSuccessHandler =
OidcClientInitiatedLogoutSuccessHandler(clientRegistrationRepository)

        // Sets the location that the End-User's User Agent will be redirected to
        // after the logout has been performed at the Provider
        oidcLogoutSuccessHandler.setPostLogoutRedirectUri("{baseUrl}")
        return oidcLogoutSuccessHandler
    }
}

NOTE: `OidcClientInitiatedLogoutSuccessHandler` supports the `{baseUrl}`
placeholder.
If used, the application's base URL, like `https://app.example.org`, will replace
it at request time.
```

12.2. OAuth 2.0 Client

The OAuth 2.0 Client features provide support for the Client role as defined in the OAuth 2.0 Authorization Framework.

At a high-level, the core features available are:

Authorization Grant support

- Authorization Code
- Refresh Token
- Client Credentials

- Resource Owner Password Credentials
- JWT Bearer

Client Authentication support

- JWT Bearer

HTTP Client support

- `WebClient` integration for Servlet Environments (for requesting protected resources)

The `HttpSecurity.oauth2Client()` DSL provides a number of configuration options for customizing the core components used by OAuth 2.0 Client. In addition, `HttpSecurity.oauth2Client().authorizationCodeGrant()` enables the customization of the Authorization Code grant.

The following code shows the complete configuration options provided by the `HttpSecurity.oauth2Client()` DSL:

Example 116. OAuth2 Client Configuration Options

Java

```
@EnableWebSecurity
public class OAuth2ClientSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Client(oauth2 -> oauth2
                .clientRegistrationRepository(this.clientRegistrationRepository())
                .authorizedClientRepository(this.authorizedClientRepository())
                .authorizedClientService(this.authorizedClientService())
                .authorizationCodeGrant(codeGrant -> codeGrant

.authorizationRequestRepository(this.authorizationRequestRepository())

.authorizationRequestResolver(this.authorizationRequestResolver())
                    .accessTokenResponseClient(this.accessTokenResponseClient())
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2ClientSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            oauth2Client {
                clientRegistrationRepository = clientRegistrationRepository()
                authorizedClientRepository = authorizedClientRepository()
                authorizedClientService = authorizedClientService()
                authorizationCodeGrant {
                    authorizationRequestRepository =
authorizationRequestRepository()
                    authorizationRequestResolver = authorizationRequestResolver()
                    accessTokenResponseClient = accessTokenResponseClient()
                }
            }
        }
    }
}
```

In addition to the `HttpSecurity.oauth2Client()` DSL, XML configuration is also supported.

The following code shows the complete configuration options available in the security namespace:

Example 117. OAuth2 Client XML Configuration Options

```
<http>
    <oauth2-client client-registration-repository-
ref="clientRegistrationRepository"
                   authorized-client-repository-ref="authorizedClientRepository"
                   authorized-client-service-ref="authorizedClientService">
        <authorization-code-grant
                authorization-request-repository-
ref="authorizationRequestRepository"
                authorization-request-resolver-ref="authorizationRequestResolver"
                access-token-response-client-ref="accessTokenResponseClient"/>
    </oauth2-client>
</http>
```

The `OAuth2AuthorizedClientManager` is responsible for managing the authorization (or re-authorization) of an OAuth 2.0 Client, in collaboration with one or more `OAuth2AuthorizedClientProvider`(s).

The following code shows an example of how to register an `OAuth2AuthorizedClientManager` `@Bean` and associate it with an `OAuth2AuthorizedClientProvider` composite that provides support for the `authorization_code`, `refresh_token`, `client_credentials` and `password` authorization grant types:

Java

```
@Bean
public OAuth2AuthorizedClientManager authorizedClientManager(
        ClientRegistrationRepository clientRegistrationRepository,
        OAuth2AuthorizedClientRepository authorizedClientRepository) {

    OAuth2AuthorizedClientProvider authorizedClientProvider =
            OAuth2AuthorizedClientProviderBuilder.builder()
                    .authorizationCode()
                    .refreshToken()
                    .clientCredentials()
                    .password()
                    .build();

    DefaultOAuth2AuthorizedClientManager authorizedClientManager =
            new DefaultOAuth2AuthorizedClientManager(
                    clientRegistrationRepository, authorizedClientRepository);
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);

    return authorizedClientManager;
}
```

Kotlin

```
@Bean
fun authorizedClientManager(
        clientRegistrationRepository: ClientRegistrationRepository,
        authorizedClientRepository: OAuth2AuthorizedClientRepository):
OAuth2AuthorizedClientManager {
    val authorizedClientProvider: OAuth2AuthorizedClientProvider =
OAuth2AuthorizedClientProviderBuilder.builder()
            .authorizationCode()
            .refreshToken()
            .clientCredentials()
            .password()
            .build()
    val authorizedClientManager = DefaultOAuth2AuthorizedClientManager(
            clientRegistrationRepository, authorizedClientRepository)
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider)
    return authorizedClientManager
}
```

The following sections will go into more detail on the core components used by OAuth 2.0 Client and the configuration options available:

- Core Interfaces / Classes
  - ClientRegistration

12.2.1. Core Interfaces / Classes

ClientRegistration

`ClientRegistration` is a representation of a client registered with an OAuth 2.0 or OpenID Connect 1.0 Provider.

A client registration holds information, such as client id, client secret, authorization grant type, redirect URI, scope(s), authorization URI, token URI, and other details.

`ClientRegistration` and its properties are defined as follows:

```
public final class ClientRegistration {
    private String registrationId;  ①
    private String clientId;    ②
    private String clientSecret;    ③
    private ClientAuthenticationMethod clientAuthenticationMethod;  ④
    private AuthorizationGrantType authorizationGrantType;  ⑤
    private String redirectUri; ⑥
    private Set<String> scopes; ⑦
    private ProviderDetails providerDetails;
    private String clientName;  ⑧

    public class ProviderDetails {
        private String authorizationUri;    ⑨
        private String tokenUri;    ⑩
        private UserInfoEndpoint userInfoEndpoint;
        private String jwkSetUri;   ⑪
        private String issuerUri;   ⑫
        private Map<String, Object> configurationMetadata;  ⑬

        public class UserInfoEndpoint {
            private String uri; ⑭
            private AuthenticationMethod authenticationMethod;  ⑮
            private String userNameAttributeName;   ⑯

        }
    }
}
```

① `registrationId`: The ID that uniquely identifies the `ClientRegistration`.

② `clientId`: The client identifier.

③ `clientSecret`: The client secret.

④ `clientAuthenticationMethod`: The method used to authenticate the Client with the Provider. The supported values are **client_secret_basic**, **client_secret_post**, **private_key_jwt**, **client_secret_jwt** and **none** (public clients).

⑤ `authorizationGrantType`: The OAuth 2.0 Authorization Framework defines four Authorization Grant types. The supported values are `authorization_code`, `client_credentials`, `password`, as well as, extension grant type `urn:ietf:params:oauth:grant-type:jwt-bearer`.

⑥ `redirectUri`: The client's registered redirect URI that the *Authorization Server* redirects the end-user's user-agent to after the end-user has authenticated and authorized access to the client.

⑦ `scopes`: The scope(s) requested by the client during the Authorization Request flow, such as openid, email, or profile.

⑧ `clientName`: A descriptive name used for the client. The name may be used in certain scenarios, such as when displaying the name of the client in the auto-generated login page.

⑨ `authorizationUri`: The Authorization Endpoint URI for the Authorization Server.

⑩ `tokenUri`: The Token Endpoint URI for the Authorization Server.

⑪ `jwkSetUri`: The URI used to retrieve the JSON Web Key (JWK) Set from the Authorization Server, which contains the cryptographic key(s) used to verify the JSON Web Signature (JWS) of the ID Token and optionally the UserInfo Response.

⑫ `issuerUri`: Returns the issuer identifier uri for the OpenID Connect 1.0 provider or the OAuth 2.0 Authorization Server.

⑬ `configurationMetadata`: The OpenID Provider Configuration Information. This information will only be available if the Spring Boot 2.x property `spring.security.oauth2.client.provider.[providerId].issuerUri` is configured.

⑭ `(userInfoEndpoint)uri`: The UserInfo Endpoint URI used to access the claims/attributes of the authenticated end-user.

⑮ `(userInfoEndpoint)authenticationMethod`: The authentication method used when sending the access token to the UserInfo Endpoint. The supported values are **header**, **form** and **query**.

⑯ `userNameAttributeName`: The name of the attribute returned in the UserInfo Response that references the Name or Identifier of the end-user.

A `ClientRegistration` can be initially configured using discovery of an OpenID Connect Provider's Configuration endpoint or an Authorization Server's Metadata endpoint.

`ClientRegistrations` provides convenience methods for configuring a `ClientRegistration` in this way, as can be seen in the following example:

Java

```
ClientRegistration clientRegistration =

ClientRegistrations.fromIssuerLocation("https://idp.example.com/issuer").build();
```

Kotlin

```
val clientRegistration =
ClientRegistrations.fromIssuerLocation("https://idp.example.com/issuer").build()
```

The above code will query in series `https://idp.example.com/issuer/.well-known/openid-configuration`, and then `https://idp.example.com/.well-known/openid-configuration/issuer`, and finally `https://idp.example.com/.well-known/oauth-authorization-server/issuer`, stopping at the first to return a 200 response.

As an alternative, you can use `ClientRegistrations.fromOidcIssuerLocation()` to only query the OpenID Connect Provider's Configuration endpoint.

ClientRegistrationRepository

The `ClientRegistrationRepository` serves as a repository for OAuth 2.0 / OpenID Connect 1.0 `ClientRegistration`(s).

Client registration information is ultimately stored and owned by the associated Authorization Server. This repository provides the ability to retrieve a sub-set of the primary client registration information, which is stored with the Authorization Server.

Spring Boot 2.x auto-configuration binds each of the properties under `spring.security.oauth2.client.registration.[registrationId]` to an instance of `ClientRegistration` and then composes each of the `ClientRegistration` instance(s) within a `ClientRegistrationRepository`.

The default implementation of `ClientRegistrationRepository` is `InMemoryClientRegistrationRepository`.

The auto-configuration also registers the `ClientRegistrationRepository` as a `@Bean` in the `ApplicationContext` so that it is available for dependency-injection, if needed by the application.

The following listing shows an example:

Java

```
@Controller
public class OAuth2ClientController {

    @Autowired
    private ClientRegistrationRepository clientRegistrationRepository;

    @GetMapping("/")
    public String index() {
        ClientRegistration oktaRegistration =
            this.clientRegistrationRepository.findByRegistrationId("okta");

        ...

        return "index";
    }
}
```

Kotlin

```
@Controller
class OAuth2ClientController {

    @Autowired
    private lateinit var clientRegistrationRepository:
ClientRegistrationRepository

    @GetMapping("/")
    fun index(): String {
        val oktaRegistration =
                this.clientRegistrationRepository.findByRegistrationId("okta")

        //...

        return "index";
    }
}
```

OAuth2AuthorizedClient

`OAuth2AuthorizedClient` is a representation of an Authorized Client. A client is considered to be authorized when the end-user (Resource Owner) has granted authorization to the client to access its protected resources.

`OAuth2AuthorizedClient` serves the purpose of associating an `OAuth2AccessToken` (and optional `OAuth2RefreshToken`) to a `ClientRegistration` (client) and resource owner, who is the `Principal` end-user that granted the authorization.

OAuth2AuthorizedClientRepository / OAuth2AuthorizedClientService

`OAuth2AuthorizedClientRepository` is responsible for persisting `OAuth2AuthorizedClient`(s) between web requests. Whereas, the primary role of `OAuth2AuthorizedClientService` is to manage `OAuth2AuthorizedClient`(s) at the application-level.

From a developer perspective, the `OAuth2AuthorizedClientRepository` or `OAuth2AuthorizedClientService` provides the capability to lookup an `OAuth2AccessToken` associated with a client so that it may be used to initiate a protected resource request.

The following listing shows an example:

Java

```
@Controller
public class OAuth2ClientController {

    @Autowired
    private OAuth2AuthorizedClientService authorizedClientService;

    @GetMapping("/")
    public String index(Authentication authentication) {
        OAuth2AuthorizedClient authorizedClient =
            this.authorizedClientService.loadAuthorizedClient("okta",
authentication.getName());

        OAuth2AccessToken accessToken = authorizedClient.getAccessToken();

        ...

        return "index";
    }
}
```

Kotlin

```
@Controller
class OAuth2ClientController {

    @Autowired
    private lateinit var authorizedClientService: OAuth2AuthorizedClientService

    @GetMapping("/")
    fun index(authentication: Authentication): String {
        val authorizedClient: OAuth2AuthorizedClient =
            this.authorizedClientService.loadAuthorizedClient("okta",
authentication.getName());
        val accessToken = authorizedClient.accessToken

        ...

        return "index";
    }
}
```

Spring Boot 2.x auto-configuration registers an `OAuth2AuthorizedClientRepository` and/or `OAuth2AuthorizedClientService` `@Bean` in the `ApplicationContext`. However, the application may choose to override and register a custom `OAuth2AuthorizedClientRepository` or `OAuth2AuthorizedClientService` `@Bean`.

The default implementation of `OAuth2AuthorizedClientService` is `InMemoryOAuth2AuthorizedClientService`, which stores `OAuth2AuthorizedClient`(s) in-memory.

Alternatively, the JDBC implementation `JdbcOAuth2AuthorizedClientService` may be configured for persisting `OAuth2AuthorizedClient`(s) in a database.

`JdbcOAuth2AuthorizedClientService` depends on the table definition described in OAuth 2.0 Client Schema.

OAuth2AuthorizedClientManager / OAuth2AuthorizedClientProvider

The `OAuth2AuthorizedClientManager` is responsible for the overall management of `OAuth2AuthorizedClient`(s).

The primary responsibilities include:

- Authorizing (or re-authorizing) an OAuth 2.0 Client, using an `OAuth2AuthorizedClientProvider`.
- Delegating the persistence of an `OAuth2AuthorizedClient`, typically using an `OAuth2AuthorizedClientService` or `OAuth2AuthorizedClientRepository`.
- Delegating to an `OAuth2AuthorizationSuccessHandler` when an OAuth 2.0 Client has been successfully authorized (or re-authorized).
- Delegating to an `OAuth2AuthorizationFailureHandler` when an OAuth 2.0 Client fails to authorize (or re-authorize).

An `OAuth2AuthorizedClientProvider` implements a strategy for authorizing (or re-authorizing) an OAuth 2.0 Client. Implementations will typically implement an authorization grant type, eg. `authorization_code`, `client_credentials`, etc.

The default implementation of `OAuth2AuthorizedClientManager` is `DefaultOAuth2AuthorizedClientManager`, which is associated with an `OAuth2AuthorizedClientProvider` that may support multiple authorization grant types using a delegation-based composite. The `OAuth2AuthorizedClientProviderBuilder` may be used to configure and build the delegation-based composite.

The following code shows an example of how to configure and build an `OAuth2AuthorizedClientProvider` composite that provides support for the `authorization_code`, `refresh_token`, `client_credentials` and `password` authorization grant types:

Java

```
@Bean
public OAuth2AuthorizedClientManager authorizedClientManager(
        ClientRegistrationRepository clientRegistrationRepository,
        OAuth2AuthorizedClientRepository authorizedClientRepository) {

    OAuth2AuthorizedClientProvider authorizedClientProvider =
            OAuth2AuthorizedClientProviderBuilder.builder()
                    .authorizationCode()
                    .refreshToken()
                    .clientCredentials()
                    .password()
                    .build();

    DefaultOAuth2AuthorizedClientManager authorizedClientManager =
            new DefaultOAuth2AuthorizedClientManager(
                    clientRegistrationRepository, authorizedClientRepository);
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);

    return authorizedClientManager;
}
```

Kotlin

```
@Bean
fun authorizedClientManager(
        clientRegistrationRepository: ClientRegistrationRepository,
        authorizedClientRepository: OAuth2AuthorizedClientRepository):
OAuth2AuthorizedClientManager {
    val authorizedClientProvider = OAuth2AuthorizedClientProviderBuilder.builder()
            .authorizationCode()
            .refreshToken()
            .clientCredentials()
            .password()
            .build()
    val authorizedClientManager = DefaultOAuth2AuthorizedClientManager(
            clientRegistrationRepository, authorizedClientRepository)
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider)
    return authorizedClientManager
}
```

When an authorization attempt succeeds, the `DefaultOAuth2AuthorizedClientManager` will delegate to the `OAuth2AuthorizationSuccessHandler`, which (by default) will save the `OAuth2AuthorizedClient` via the `OAuth2AuthorizedClientRepository`. In the case of a re-authorization failure, eg. a refresh token is no longer valid, the previously saved `OAuth2AuthorizedClient` will be removed from the `OAuth2AuthorizedClientRepository` via the `RemoveAuthorizedClientOAuth2AuthorizationFailureHandler`. The default behaviour may be

customized via `setAuthorizationSuccessHandler(OAuth2AuthorizationSuccessHandler)` and `setAuthorizationFailureHandler(OAuth2AuthorizationFailureHandler)`.

The `DefaultOAuth2AuthorizedClientManager` is also associated with a `contextAttributesMapper` of type `Function<OAuth2AuthorizeRequest, Map<String, Object>>`, which is responsible for mapping attribute(s) from the `OAuth2AuthorizeRequest` to a `Map` of attributes to be associated to the `OAuth2AuthorizationContext`. This can be useful when you need to supply an `OAuth2AuthorizedClientProvider` with required (supported) attribute(s), eg. the `PasswordOAuth2AuthorizedClientProvider` requires the resource owner's `username` and `password` to be available in `OAuth2AuthorizationContext.getAttributes()`.

The following code shows an example of the `contextAttributesMapper`:

Java

```
@Bean
public OAuth2AuthorizedClientManager authorizedClientManager(
        ClientRegistrationRepository clientRegistrationRepository,
        OAuth2AuthorizedClientRepository authorizedClientRepository) {

    OAuth2AuthorizedClientProvider authorizedClientProvider =
            OAuth2AuthorizedClientProviderBuilder.builder()
                    .password()
                    .refreshToken()
                    .build();

    DefaultOAuth2AuthorizedClientManager authorizedClientManager =
            new DefaultOAuth2AuthorizedClientManager(
                    clientRegistrationRepository, authorizedClientRepository);
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);

    // Assuming the `username` and `password` are supplied as `HttpServletRequest`
parameters,
    // map the `HttpServletRequest` parameters to
`OAuth2AuthorizationContext.getAttributes()`
    authorizedClientManager.setContextAttributesMapper(contextAttributesMapper());

    return authorizedClientManager;
}

private Function<OAuth2AuthorizeRequest, Map<String, Object>>
contextAttributesMapper() {
    return authorizeRequest -> {
        Map<String, Object> contextAttributes = Collections.emptyMap();
        HttpServletRequest servletRequest =
authorizeRequest.getAttribute(HttpServletRequest.class.getName());
        String username =
servletRequest.getParameter(OAuth2ParameterNames.USERNAME);
        String password =
servletRequest.getParameter(OAuth2ParameterNames.PASSWORD);
        if (StringUtils.hasText(username) && StringUtils.hasText(password)) {
            contextAttributes = new HashMap<>();

            // `PasswordOAuth2AuthorizedClientProvider` requires both attributes

contextAttributes.put(OAuth2AuthorizationContext.USERNAME_ATTRIBUTE_NAME,
username);

contextAttributes.put(OAuth2AuthorizationContext.PASSWORD_ATTRIBUTE_NAME,
password);
        }
        return contextAttributes;
    };
}
```

Kotlin

```
@Bean
fun authorizedClientManager(
        clientRegistrationRepository: ClientRegistrationRepository,
        authorizedClientRepository: OAuth2AuthorizedClientRepository):
OAuth2AuthorizedClientManager {
    val authorizedClientProvider = OAuth2AuthorizedClientProviderBuilder.builder()
            .password()
            .refreshToken()
            .build()
    val authorizedClientManager = DefaultOAuth2AuthorizedClientManager(
            clientRegistrationRepository, authorizedClientRepository)
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider)

    // Assuming the `username` and `password` are supplied as `HttpServletRequest`
parameters,
    // map the `HttpServletRequest` parameters to
`OAuth2AuthorizationContext.getAttributes()`
    authorizedClientManager.setContextAttributesMapper(contextAttributesMapper())
    return authorizedClientManager
}

private fun contextAttributesMapper(): Function<OAuth2AuthorizeRequest,
MutableMap<String, Any>> {
    return Function { authorizeRequest ->
        var contextAttributes: MutableMap<String, Any> = mutableMapOf()
        val servletRequest: HttpServletRequest =
authorizeRequest.getAttribute(HttpServletRequest::class.java.name)
        val username: String =
servletRequest.getParameter(OAuth2ParameterNames.USERNAME)
        val password: String =
servletRequest.getParameter(OAuth2ParameterNames.PASSWORD)
        if (StringUtils.hasText(username) && StringUtils.hasText(password)) {
            contextAttributes = hashMapOf()

            // `PasswordOAuth2AuthorizedClientProvider` requires both attributes
            contextAttributes[OAuth2AuthorizationContext.USERNAME_ATTRIBUTE_NAME]
= username
            contextAttributes[OAuth2AuthorizationContext.PASSWORD_ATTRIBUTE_NAME]
= password
        }
        contextAttributes
    }
}
```

The `DefaultOAuth2AuthorizedClientManager` is designed to be used ***within*** the context of a `HttpServletRequest`. When operating ***outside*** of a `HttpServletRequest` context, use `AuthorizedClientServiceOAuth2AuthorizedClientManager` instead.

A *service application* is a common use case for when to use an `AuthorizedClientServiceOAuth2AuthorizedClientManager`. Service applications often run in the background, without any user interaction, and typically run under a system-level account instead of a user account. An OAuth 2.0 Client configured with the `client_credentials` grant type can be considered a type of service application.

The following code shows an example of how to configure an `AuthorizedClientServiceOAuth2AuthorizedClientManager` that provides support for the `client_credentials` grant type:

Java

```
@Bean
public OAuth2AuthorizedClientManager authorizedClientManager(
        ClientRegistrationRepository clientRegistrationRepository,
        OAuth2AuthorizedClientService authorizedClientService) {

    OAuth2AuthorizedClientProvider authorizedClientProvider =
            OAuth2AuthorizedClientProviderBuilder.builder()
                    .clientCredentials()
                    .build();

    AuthorizedClientServiceOAuth2AuthorizedClientManager authorizedClientManager =
            new AuthorizedClientServiceOAuth2AuthorizedClientManager(
                    clientRegistrationRepository, authorizedClientService);
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);

    return authorizedClientManager;
}
```

Kotlin

```
@Bean
fun authorizedClientManager(
        clientRegistrationRepository: ClientRegistrationRepository,
        authorizedClientService: OAuth2AuthorizedClientService):
OAuth2AuthorizedClientManager {
    val authorizedClientProvider = OAuth2AuthorizedClientProviderBuilder.builder()
            .clientCredentials()
            .build()
    val authorizedClientManager =
AuthorizedClientServiceOAuth2AuthorizedClientManager(
            clientRegistrationRepository, authorizedClientService)
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider)
    return authorizedClientManager
}
```

12.2.2. Authorization Grant Support

Authorization Code

Please refer to the OAuth 2.0 Authorization Framework for further details on the Authorization Code grant.

Obtaining Authorization

Please refer to the Authorization Request/Response protocol flow for the Authorization Code grant.

Initiating the Authorization Request

The `OAuth2AuthorizationRequestRedirectFilter` uses an `OAuth2AuthorizationRequestResolver` to resolve an `OAuth2AuthorizationRequest` and initiate the Authorization Code grant flow by redirecting the end-user's user-agent to the Authorization Server's Authorization Endpoint.

The primary role of the `OAuth2AuthorizationRequestResolver` is to resolve an `OAuth2AuthorizationRequest` from the provided web request. The default implementation `DefaultOAuth2AuthorizationRequestResolver` matches on the (default) path `/oauth2/authorization/{registrationId}` extracting the `registrationId` and using it to build the `OAuth2AuthorizationRequest` for the associated `ClientRegistration`.

Given the following Spring Boot 2.x properties for an OAuth 2.0 Client registration:

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: okta-client-id
            client-secret: okta-client-secret
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/authorized/okta"
            scope: read, write
        provider:
          okta:
            authorization-uri: https://dev-1234.oktapreview.com/oauth2/v1/authorize
            token-uri: https://dev-1234.oktapreview.com/oauth2/v1/token
```

A request with the base path `/oauth2/authorization/okta` will initiate the Authorization Request redirect by the `OAuth2AuthorizationRequestRedirectFilter` and ultimately start the Authorization Code grant flow.

The `AuthorizationCodeOAuth2AuthorizedClientProvider` is an implementation of `OAuth2AuthorizedClientProvider` for the Authorization Code grant, which also initiates the Authorization Request redirect by the `OAuth2AuthorizationRequestRedirectFilter`.

If the OAuth 2.0 Client is a Public Client, then configure the OAuth 2.0 Client registration as follows:

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: okta-client-id
            client-authentication-method: none
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/authorized/okta"
            ...
```

Public Clients are supported using Proof Key for Code Exchange (PKCE). If the client is running in an untrusted environment (eg. native application or web browser-based application) and therefore incapable of maintaining the confidentiality of it's credentials, PKCE will automatically be used when the following conditions are true:

1. `client-secret` is omitted (or empty)
2. `client-authentication-method` is set to "none" (`ClientAuthenticationMethod.NONE`)

The `DefaultOAuth2AuthorizationRequestResolver` also supports `URI` template variables for the `redirect-uri` using `UriComponentsBuilder`.

The following configuration uses all the supported `URI` template variables:

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            ...
            redirect-uri:
"{baseScheme}://{baseHost}{basePort}{basePath}/authorized/{registrationId}"
            ...
```

`{baseUrl}` resolves to `{baseScheme}://{baseHost}{basePort}{basePath}`

Configuring the `redirect-uri` with `URI` template variables is especially useful when the OAuth 2.0 Client is running behind a Proxy Server. This ensures that the `X-Forwarded-*` headers are used when

expanding the `redirect-uri`.

Customizing the Authorization Request

One of the primary use cases an `OAuth2AuthorizationRequestResolver` can realize is the ability to customize the Authorization Request with additional parameters above the standard parameters defined in the OAuth 2.0 Authorization Framework.

For example, OpenID Connect defines additional OAuth 2.0 request parameters for the Authorization Code Flow extending from the standard parameters defined in the OAuth 2.0 Authorization Framework. One of those extended parameters is the `prompt` parameter.

> OPTIONAL. Space delimited, case sensitive list of ASCII string values that specifies whether the Authorization Server prompts the End-User for reauthentication and consent. The defined values are: none, login, consent, select_account

The following example shows how to configure the `DefaultOAuth2AuthorizationRequestResolver` with a `Consumer<OAuth2AuthorizationRequest.Builder>` that customizes the Authorization Request for `oauth2Login()`, by including the request parameter `prompt=consent`.

Java

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private ClientRegistrationRepository clientRegistrationRepository;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .oauth2Login(oauth2 -> oauth2
                .authorizationEndpoint(authorization -> authorization
                    .authorizationRequestResolver(

authorizationRequestResolver(this.clientRegistrationRepository)
                    )
                )
            );
    }

    private OAuth2AuthorizationRequestResolver authorizationRequestResolver(
            ClientRegistrationRepository clientRegistrationRepository) {

        DefaultOAuth2AuthorizationRequestResolver authorizationRequestResolver =
                new DefaultOAuth2AuthorizationRequestResolver(
                        clientRegistrationRepository, "/oauth2/authorization");
        authorizationRequestResolver.setAuthorizationRequestCustomizer(
                authorizationRequestCustomizer());

        return  authorizationRequestResolver;
    }

    private Consumer<OAuth2AuthorizationRequest.Builder>
authorizationRequestCustomizer() {
        return customizer -> customizer
                    .additionalParameters(params -> params.put("prompt",
"consent"));
    }
}
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    @Autowired
    private lateinit var customClientRegistrationRepository:
ClientRegistrationRepository

    override fun configure(http: HttpSecurity) {
        http {
            authorizeRequests {
                authorize(anyRequest, authenticated)
            }
            oauth2Login {
                authorizationEndpoint {
                    authorizationRequestResolver =
authorizationRequestResolver(customClientRegistrationRepository)
                }
            }
        }
    }

    private fun authorizationRequestResolver(
            clientRegistrationRepository: ClientRegistrationRepository?):
OAuth2AuthorizationRequestResolver? {
        val authorizationRequestResolver =
DefaultOAuth2AuthorizationRequestResolver(
                clientRegistrationRepository, "/oauth2/authorization")
        authorizationRequestResolver.setAuthorizationRequestCustomizer(
                authorizationRequestCustomizer())
        return authorizationRequestResolver
    }

    private fun authorizationRequestCustomizer():
Consumer<OAuth2AuthorizationRequest.Builder> {
        return Consumer { customizer ->
            customizer
                    .additionalParameters { params -> params["prompt"] = "consent"
}
        }
    }
}
```

For the simple use case, where the additional request parameter is always the same for a specific provider, it may be added directly in the `authorization-uri` property.

For example, if the value for the request parameter `prompt` is always `consent` for the provider `okta`, than simply configure as follows:

```
spring:
  security:
    oauth2:
      client:
        provider:
          okta:
            authorization-uri: https://dev-
1234.oktapreview.com/oauth2/v1/authorize?prompt=consent
```

The preceding example shows the common use case of adding a custom parameter on top of the standard parameters. Alternatively, if your requirements are more advanced, you can take full control in building the Authorization Request URI by simply overriding the `OAuth2AuthorizationRequest.authorizationRequestUri` property.

`OAuth2AuthorizationRequest.Builder.build()` constructs the `OAuth2AuthorizationRequest.authorizationRequestUri`, which represents the Authorization Request URI including all query parameters using the `application/x-www-form-urlencoded` format.

The following example shows a variation of `authorizationRequestCustomizer()` from the preceding example, and instead overrides the `OAuth2AuthorizationRequest.authorizationRequestUri` property.

Java

```
private Consumer<OAuth2AuthorizationRequest.Builder>
authorizationRequestCustomizer() {
    return customizer -> customizer
                .authorizationRequestUri(uriBuilder -> uriBuilder
                    .queryParam("prompt", "consent").build());
}
```

Kotlin

```
private fun authorizationRequestCustomizer():
Consumer<OAuth2AuthorizationRequest.Builder> {
    return Consumer { customizer: OAuth2AuthorizationRequest.Builder ->
        customizer
                .authorizationRequestUri { uriBuilder: UriBuilder ->
                    uriBuilder
                            .queryParam("prompt", "consent").build()
                }
    }
}
```

Storing the Authorization Request

The `AuthorizationRequestRepository` is responsible for the persistence of the `OAuth2AuthorizationRequest` from the time the Authorization Request is initiated to the time the Authorization Response is received (the callback).

The `OAuth2AuthorizationRequest` is used to correlate and validate the Authorization Response.

The default implementation of `AuthorizationRequestRepository` is `HttpSessionOAuth2AuthorizationRequestRepository`, which stores the `OAuth2AuthorizationRequest` in the `HttpSession`.

If you have a custom implementation of `AuthorizationRequestRepository`, you may configure it as shown in the following example:

Example 118. AuthorizationRequestRepository Configuration

Java

```
@EnableWebSecurity
public class OAuth2ClientSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Client(oauth2 -> oauth2
                .authorizationCodeGrant(codeGrant -> codeGrant

.authorizationRequestRepository(this.authorizationRequestRepository())
                    ...
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2ClientSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            oauth2Client {
                authorizationCodeGrant {
                    authorizationRequestRepository =
authorizationRequestRepository()
                }
            }
        }
    }
}
```

Xml

```
<http>
    <oauth2-client>
        <authorization-code-grant authorization-request-repository-
ref="authorizationRequestRepository"/>
    </oauth2-client>
</http>
```

Requesting an Access Token

Please refer to the Access Token Request/Response protocol flow for the Authorization Code grant.

The default implementation of `OAuth2AccessTokenResponseClient` for the Authorization Code grant is `DefaultAuthorizationCodeTokenResponseClient`, which uses a `RestOperations` for exchanging an authorization code for an access token at the Authorization Server's Token Endpoint.

The `DefaultAuthorizationCodeTokenResponseClient` is quite flexible as it allows you to customize the pre-processing of the Token Request and/or post-handling of the Token Response.

Customizing the Access Token Request

If you need to customize the pre-processing of the Token Request, you can provide `DefaultAuthorizationCodeTokenResponseClient.setRequestEntityConverter()` with a custom `Converter<OAuth2AuthorizationCodeGrantRequest, RequestEntity<?>>`. The default implementation `OAuth2AuthorizationCodeGrantRequestEntityConverter` builds a `RequestEntity` representation of a standard OAuth 2.0 Access Token Request. However, providing a custom `Converter`, would allow you to extend the standard Token Request and add custom parameter(s).

The custom `Converter` must return a valid `RequestEntity` representation of an OAuth 2.0 Access Token Request that is understood by the intended OAuth 2.0 Provider.

Customizing the Access Token Response

On the other end, if you need to customize the post-handling of the Token Response, you will need to provide `DefaultAuthorizationCodeTokenResponseClient.setRestOperations()` with a custom configured `RestOperations`. The default `RestOperations` is configured as follows:

Java

```
RestTemplate restTemplate = new RestTemplate(Arrays.asList(
        new FormHttpMessageConverter(),
        new OAuth2AccessTokenResponseHttpMessageConverter()));

restTemplate.setErrorHandler(new OAuth2ErrorResponseErrorHandler());
```

Kotlin

```
val restTemplate = RestTemplate(listOf(
        FormHttpMessageConverter(),
        OAuth2AccessTokenResponseHttpMessageConverter()))

restTemplate.errorHandler = OAuth2ErrorResponseErrorHandler()
```

Spring MVC `FormHttpMessageConverter` is required as it's used when sending the OAuth 2.0 Access Token Request.

`OAuth2AccessTokenResponseHttpMessageConverter` is a `HttpMessageConverter` for an OAuth 2.0 Access Token Response. You can provide `OAuth2AccessTokenResponseHttpMessageConverter.setTokenResponseConverter()` with a custom `Converter<Map<String, String>, OAuth2AccessTokenResponse>` that is used for converting the OAuth 2.0 Access Token Response parameters to an `OAuth2AccessTokenResponse`.

`OAuth2ErrorResponseErrorHandler` is a `ResponseErrorHandler` that can handle an OAuth 2.0 Error, eg. 400 Bad Request. It uses an `OAuth2ErrorHttpMessageConverter` for converting the OAuth 2.0 Error parameters to an `OAuth2Error`.

Whether you customize `DefaultAuthorizationCodeTokenResponseClient` or provide your own implementation of `OAuth2AccessTokenResponseClient`, you'll need to configure it as shown in the following example:

Example 119. Access Token Response Configuration

Java

```
@EnableWebSecurity
public class OAuth2ClientSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Client(oauth2 -> oauth2
                .authorizationCodeGrant(codeGrant -> codeGrant
                    .accessTokenResponseClient(this.accessTokenResponseClient())
                    ...
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class OAuth2ClientSecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            oauth2Client {
                authorizationCodeGrant {
                    accessTokenResponseClient = accessTokenResponseClient()
                }
            }
        }
    }
}
```

Xml

```
<http>
    <oauth2-client>
        <authorization-code-grant access-token-response-client-
ref="accessTokenResponseClient"/>
    </oauth2-client>
</http>
```

Refresh Token

Please refer to the OAuth 2.0 Authorization Framework for further details on the Refresh Token.

Refreshing an Access Token

Please refer to the Access Token Request/Response protocol flow for the Refresh Token grant.

The default implementation of `OAuth2AccessTokenResponseClient` for the Refresh Token grant is `DefaultRefreshTokenTokenResponseClient`, which uses a `RestOperations` when refreshing an access token at the Authorization Server's Token Endpoint.

The `DefaultRefreshTokenTokenResponseClient` is quite flexible as it allows you to customize the pre-processing of the Token Request and/or post-handling of the Token Response.

Customizing the Access Token Request

If you need to customize the pre-processing of the Token Request, you can provide `DefaultRefreshTokenTokenResponseClient.setRequestEntityConverter()` with a custom `Converter<OAuth2RefreshTokenGrantRequest, RequestEntity<?>>`. The default implementation `OAuth2RefreshTokenGrantRequestEntityConverter` builds a `RequestEntity` representation of a standard OAuth 2.0 Access Token Request. However, providing a custom `Converter`, would allow you to extend the standard Token Request and add custom parameter(s).

The custom `Converter` must return a valid `RequestEntity` representation of an OAuth 2.0 Access Token Request that is understood by the intended OAuth 2.0 Provider.

Customizing the Access Token Response

On the other end, if you need to customize the post-handling of the Token Response, you will need to provide `DefaultRefreshTokenTokenResponseClient.setRestOperations()` with a custom configured `RestOperations`. The default `RestOperations` is configured as follows:

Java

```
RestTemplate restTemplate = new RestTemplate(Arrays.asList(
        new FormHttpMessageConverter(),
        new OAuth2AccessTokenResponseHttpMessageConverter()));

restTemplate.setErrorHandler(new OAuth2ErrorResponseErrorHandler());
```

Kotlin

```
val restTemplate = RestTemplate(listOf(
        FormHttpMessageConverter(),
        OAuth2AccessTokenResponseHttpMessageConverter()))

restTemplate.errorHandler = OAuth2ErrorResponseErrorHandler()
```

Spring MVC `FormHttpMessageConverter` is required as it's used when sending the OAuth 2.0 Access Token Request.

`OAuth2AccessTokenResponseHttpMessageConverter` is a `HttpMessageConverter` for an OAuth 2.0 Access Token Response. You can provide `OAuth2AccessTokenResponseHttpMessageConverter.setTokenResponseConverter()` with a custom `Converter<Map<String, String>, OAuth2AccessTokenResponse>` that is used for converting the OAuth 2.0 Access Token Response parameters to an `OAuth2AccessTokenResponse`.

`OAuth2ErrorResponseErrorHandler` is a `ResponseErrorHandler` that can handle an OAuth 2.0 Error, eg. 400 Bad Request. It uses an `OAuth2ErrorHttpMessageConverter` for converting the OAuth 2.0 Error parameters to an `OAuth2Error`.

Whether you customize `DefaultRefreshTokenTokenResponseClient` or provide your own implementation of `OAuth2AccessTokenResponseClient`, you'll need to configure it as shown in the following example:

Java

```
// Customize
OAuth2AccessTokenResponseClient<OAuth2RefreshTokenGrantRequest>
refreshTokenTokenResponseClient = ...

OAuth2AuthorizedClientProvider authorizedClientProvider =
        OAuth2AuthorizedClientProviderBuilder.builder()
                .authorizationCode()
                .refreshToken(configurer ->
configurer.accessTokenResponseClient(refreshTokenTokenResponseClient))
                .build();

...

authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);
```

Kotlin

```
// Customize
val refreshTokenTokenResponseClient:
OAuth2AccessTokenResponseClient<OAuth2RefreshTokenGrantRequest> = ...

val authorizedClientProvider = OAuth2AuthorizedClientProviderBuilder.builder()
        .authorizationCode()
        .refreshToken {
it.accessTokenResponseClient(refreshTokenTokenResponseClient) }
        .build()

...

authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider)
```

`OAuth2AuthorizedClientProviderBuilder.builder().refreshToken()` configures a `RefreshTokenOAuth2AuthorizedClientProvider`, which is an implementation of an `OAuth2AuthorizedClientProvider` for the Refresh Token grant.

The `OAuth2RefreshToken` may optionally be returned in the Access Token Response for the `authorization_code` and `password` grant types. If the `OAuth2AuthorizedClient.getRefreshToken()` is available and the `OAuth2AuthorizedClient.getAccessToken()` is expired, it will automatically be refreshed by the `RefreshTokenOAuth2AuthorizedClientProvider`.

Client Credentials

Please refer to the OAuth 2.0 Authorization Framework for further details on the Client Credentials grant.

Requesting an Access Token

Please refer to the Access Token Request/Response protocol flow for the Client Credentials grant.

The default implementation of `OAuth2AccessTokenResponseClient` for the Client Credentials grant is `DefaultClientCredentialsTokenResponseClient`, which uses a `RestOperations` when requesting an access token at the Authorization Server's Token Endpoint.

The `DefaultClientCredentialsTokenResponseClient` is quite flexible as it allows you to customize the pre-processing of the Token Request and/or post-handling of the Token Response.

Customizing the Access Token Request

If you need to customize the pre-processing of the Token Request, you can provide `DefaultClientCredentialsTokenResponseClient.setRequestEntityConverter()` with a custom `Converter<OAuth2ClientCredentialsGrantRequest, RequestEntity<?>>`. The default implementation `OAuth2ClientCredentialsGrantRequestEntityConverter` builds a `RequestEntity` representation of a standard OAuth 2.0 Access Token Request. However, providing a custom `Converter`, would allow you to extend the standard Token Request and add custom parameter(s).

The custom `Converter` must return a valid `RequestEntity` representation of an OAuth 2.0 Access Token Request that is understood by the intended OAuth 2.0 Provider.

Customizing the Access Token Response

On the other end, if you need to customize the post-handling of the Token Response, you will need to provide `DefaultClientCredentialsTokenResponseClient.setRestOperations()` with a custom configured `RestOperations`. The default `RestOperations` is configured as follows:

Java

```
RestTemplate restTemplate = new RestTemplate(Arrays.asList(
        new FormHttpMessageConverter(),
        new OAuth2AccessTokenResponseHttpMessageConverter()));

restTemplate.setErrorHandler(new OAuth2ErrorResponseErrorHandler());
```

Kotlin

```
val restTemplate = RestTemplate(listOf(
        FormHttpMessageConverter(),
        OAuth2AccessTokenResponseHttpMessageConverter()))

restTemplate.errorHandler = OAuth2ErrorResponseErrorHandler()
```

Spring MVC `FormHttpMessageConverter` is required as it's used when sending the OAuth 2.0 Access Token Request.

`OAuth2AccessTokenResponseHttpMessageConverter` is a `HttpMessageConverter` for an OAuth 2.0 Access Token Response. You can provide `OAuth2AccessTokenResponseHttpMessageConverter.setTokenResponseConverter()` with a custom `Converter<Map<String, String>, OAuth2AccessTokenResponse>` that is used for converting the OAuth 2.0 Access Token Response parameters to an `OAuth2AccessTokenResponse`.

`OAuth2ErrorResponseErrorHandler` is a `ResponseErrorHandler` that can handle an OAuth 2.0 Error, eg. 400 Bad Request. It uses an `OAuth2ErrorHttpMessageConverter` for converting the OAuth 2.0 Error parameters to an `OAuth2Error`.

Whether you customize `DefaultClientCredentialsTokenResponseClient` or provide your own implementation of `OAuth2AccessTokenResponseClient`, you'll need to configure it as shown in the following example:

Java

```
// Customize
OAuth2AccessTokenResponseClient<OAuth2ClientCredentialsGrantRequest>
clientCredentialsTokenResponseClient = ...

OAuth2AuthorizedClientProvider authorizedClientProvider =
        OAuth2AuthorizedClientProviderBuilder.builder()
                .clientCredentials(configurer ->
configurer.accessTokenResponseClient(clientCredentialsTokenResponseClient))
                .build();

...

authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);
```

Kotlin

```
// Customize
val clientCredentialsTokenResponseClient:
OAuth2AccessTokenResponseClient<OAuth2ClientCredentialsGrantRequest> = ...

val authorizedClientProvider = OAuth2AuthorizedClientProviderBuilder.builder()
        .clientCredentials {
it.accessTokenResponseClient(clientCredentialsTokenResponseClient) }
        .build()

...

authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider)
```

`OAuth2AuthorizedClientProviderBuilder.builder().clientCredentials()` configures a `ClientCredentialsOAuth2AuthorizedClientProvider`, which is an implementation of an `OAuth2AuthorizedClientProvider` for the Client Credentials grant.

Using the Access Token

Given the following Spring Boot 2.x properties for an OAuth 2.0 Client registration:

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: okta-client-id
            client-secret: okta-client-secret
            authorization-grant-type: client_credentials
            scope: read, write
        provider:
          okta:
            token-uri: https://dev-1234.oktapreview.com/oauth2/v1/token
```

...and the `OAuth2AuthorizedClientManager` `@Bean`:

Java

```
@Bean
public OAuth2AuthorizedClientManager authorizedClientManager(
        ClientRegistrationRepository clientRegistrationRepository,
        OAuth2AuthorizedClientRepository authorizedClientRepository) {

    OAuth2AuthorizedClientProvider authorizedClientProvider =
            OAuth2AuthorizedClientProviderBuilder.builder()
                    .clientCredentials()
                    .build();

    DefaultOAuth2AuthorizedClientManager authorizedClientManager =
            new DefaultOAuth2AuthorizedClientManager(
                    clientRegistrationRepository, authorizedClientRepository);
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);

    return authorizedClientManager;
}
```

Kotlin

```
@Bean
fun authorizedClientManager(
        clientRegistrationRepository: ClientRegistrationRepository,
        authorizedClientRepository: OAuth2AuthorizedClientRepository):
OAuth2AuthorizedClientManager {
    val authorizedClientProvider = OAuth2AuthorizedClientProviderBuilder.builder()
            .clientCredentials()
            .build()
    val authorizedClientManager = DefaultOAuth2AuthorizedClientManager(
            clientRegistrationRepository, authorizedClientRepository)
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider)
    return authorizedClientManager
}
```

You may obtain the `OAuth2AccessToken` as follows:

Java

```
@Controller
public class OAuth2ClientController {

    @Autowired
    private OAuth2AuthorizedClientManager authorizedClientManager;

    @GetMapping("/")
    public String index(Authentication authentication,
                        HttpServletRequest servletRequest,
                        HttpServletResponse servletResponse) {

        OAuth2AuthorizeRequest authorizeRequest =
OAuth2AuthorizeRequest.withClientRegistrationId("okta")
                .principal(authentication)
                .attributes(attrs -> {
                    attrs.put(HttpServletRequest.class.getName(), servletRequest);
                    attrs.put(HttpServletResponse.class.getName(),
servletResponse);
                })
                .build();
        OAuth2AuthorizedClient authorizedClient =
this.authorizedClientManager.authorize(authorizeRequest);

        OAuth2AccessToken accessToken = authorizedClient.getAccessToken();

        ...

        return "index";
    }
}
```

Kotlin

```
class OAuth2ClientController {

    @Autowired
    private lateinit var authorizedClientManager: OAuth2AuthorizedClientManager

    @GetMapping("/")
    fun index(authentication: Authentication?,
              servletRequest: HttpServletRequest,
              servletResponse: HttpServletResponse): String {
        val authorizeRequest: OAuth2AuthorizeRequest =
OAuth2AuthorizeRequest.withClientRegistrationId("okta")
                .principal(authentication)
                .attributes(Consumer { attrs: MutableMap<String, Any> ->
                    attrs[HttpServletRequest::class.java.name] = servletRequest
                    attrs[HttpServletResponse::class.java.name] = servletResponse
                })
                .build()
        val authorizedClient = authorizedClientManager.authorize(authorizeRequest)
        val accessToken: OAuth2AccessToken = authorizedClient.accessToken

        ...

        return "index"
    }
}
```

`HttpServletRequest` and `HttpServletResponse` are both OPTIONAL attributes. If not provided, it will default to `ServletRequestAttributes` using `RequestContextHolder.getRequestAttributes()`.

Resource Owner Password Credentials

Please refer to the OAuth 2.0 Authorization Framework for further details on the Resource Owner Password Credentials grant.

Requesting an Access Token

Please refer to the Access Token Request/Response protocol flow for the Resource Owner Password Credentials grant.

The default implementation of `OAuth2AccessTokenResponseClient` for the Resource Owner Password Credentials grant is `DefaultPasswordTokenResponseClient`, which uses a `RestOperations` when requesting an access token at the Authorization Server's Token Endpoint.

The `DefaultPasswordTokenResponseClient` is quite flexible as it allows you to customize the pre-processing of the Token Request and/or post-handling of the Token Response.

Customizing the Access Token Request

If you need to customize the pre-processing of the Token Request, you can provide `DefaultPasswordTokenResponseClient.setRequestEntityConverter()` with a custom `Converter<OAuth2PasswordGrantRequest, RequestEntity<?>>`. The default implementation `OAuth2PasswordGrantRequestEntityConverter` builds a `RequestEntity` representation of a standard OAuth 2.0 Access Token Request. However, providing a custom `Converter`, would allow you to extend the standard Token Request and add custom parameter(s).

The custom `Converter` must return a valid `RequestEntity` representation of an OAuth 2.0 Access Token Request that is understood by the intended OAuth 2.0 Provider.

Customizing the Access Token Response

On the other end, if you need to customize the post-handling of the Token Response, you will need to provide `DefaultPasswordTokenResponseClient.setRestOperations()` with a custom configured `RestOperations`. The default `RestOperations` is configured as follows:

Java

```
RestTemplate restTemplate = new RestTemplate(Arrays.asList(
        new FormHttpMessageConverter(),
        new OAuth2AccessTokenResponseHttpMessageConverter()));

restTemplate.setErrorHandler(new OAuth2ErrorResponseErrorHandler());
```

Kotlin

```
val restTemplate = RestTemplate(listOf(
        FormHttpMessageConverter(),
        OAuth2AccessTokenResponseHttpMessageConverter()))

restTemplate.errorHandler = OAuth2ErrorResponseErrorHandler()
```

Spring MVC `FormHttpMessageConverter` is required as it's used when sending the OAuth 2.0 Access Token Request.

`OAuth2AccessTokenResponseHttpMessageConverter` is a `HttpMessageConverter` for an OAuth 2.0 Access Token Response. You can provide `OAuth2AccessTokenResponseHttpMessageConverter.setTokenResponseConverter()` with a custom `Converter<Map<String, String>, OAuth2AccessTokenResponse>` that is used for converting the OAuth 2.0 Access Token Response parameters to an `OAuth2AccessTokenResponse`.

`OAuth2ErrorResponseErrorHandler` is a `ResponseErrorHandler` that can handle an OAuth 2.0 Error, eg. 400 Bad Request. It uses an `OAuth2ErrorHttpMessageConverter` for converting the OAuth 2.0 Error parameters to an `OAuth2Error`.

Whether you customize `DefaultPasswordTokenResponseClient` or provide your own implementation of `OAuth2AccessTokenResponseClient`, you'll need to configure it as shown in the following example:

Java

```
// Customize
OAuth2AccessTokenResponseClient<OAuth2PasswordGrantRequest>
passwordTokenResponseClient = ...

OAuth2AuthorizedClientProvider authorizedClientProvider =
        OAuth2AuthorizedClientProviderBuilder.builder()
                .password(configurer ->
configurer.accessTokenResponseClient(passwordTokenResponseClient))
                .refreshToken()
                .build();

...

authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);
```

Kotlin

```
val passwordTokenResponseClient:
OAuth2AccessTokenResponseClient<OAuth2PasswordGrantRequest> = ...

val authorizedClientProvider = OAuth2AuthorizedClientProviderBuilder.builder()
        .password { it.accessTokenResponseClient(passwordTokenResponseClient) }
        .refreshToken()
        .build()

...

authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider)
```

`OAuth2AuthorizedClientProviderBuilder.builder().password()` configures a `PasswordOAuth2AuthorizedClientProvider`, which is an implementation of an `OAuth2AuthorizedClientProvider` for the Resource Owner Password Credentials grant.

Using the Access Token

Given the following Spring Boot 2.x properties for an OAuth 2.0 Client registration:

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: okta-client-id
            client-secret: okta-client-secret
            authorization-grant-type: password
            scope: read, write
        provider:
          okta:
            token-uri: https://dev-1234.oktapreview.com/oauth2/v1/token
```

...and the `OAuth2AuthorizedClientManager` `@Bean`:

Java

```
@Bean
public OAuth2AuthorizedClientManager authorizedClientManager(
        ClientRegistrationRepository clientRegistrationRepository,
        OAuth2AuthorizedClientRepository authorizedClientRepository) {

    OAuth2AuthorizedClientProvider authorizedClientProvider =
            OAuth2AuthorizedClientProviderBuilder.builder()
                    .password()
                    .refreshToken()
                    .build();

    DefaultOAuth2AuthorizedClientManager authorizedClientManager =
            new DefaultOAuth2AuthorizedClientManager(
                    clientRegistrationRepository, authorizedClientRepository);
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);

    // Assuming the `username` and `password` are supplied as `HttpServletRequest`
parameters,
    // map the `HttpServletRequest` parameters to
`OAuth2AuthorizationContext.getAttributes()`
    authorizedClientManager.setContextAttributesMapper(contextAttributesMapper());

    return authorizedClientManager;
}

private Function<OAuth2AuthorizeRequest, Map<String, Object>>
contextAttributesMapper() {
    return authorizeRequest -> {
        Map<String, Object> contextAttributes = Collections.emptyMap();
        HttpServletRequest servletRequest =
authorizeRequest.getAttribute(HttpServletRequest.class.getName());
        String username =
servletRequest.getParameter(OAuth2ParameterNames.USERNAME);
        String password =
servletRequest.getParameter(OAuth2ParameterNames.PASSWORD);
        if (StringUtils.hasText(username) && StringUtils.hasText(password)) {
            contextAttributes = new HashMap<>();

            // `PasswordOAuth2AuthorizedClientProvider` requires both attributes

contextAttributes.put(OAuth2AuthorizationContext.USERNAME_ATTRIBUTE_NAME,
username);

contextAttributes.put(OAuth2AuthorizationContext.PASSWORD_ATTRIBUTE_NAME,
password);
        }
        return contextAttributes;
    };
}
```

Kotlin

```
@Bean
fun authorizedClientManager(
        clientRegistrationRepository: ClientRegistrationRepository,
        authorizedClientRepository: OAuth2AuthorizedClientRepository):
OAuth2AuthorizedClientManager {
    val authorizedClientProvider = OAuth2AuthorizedClientProviderBuilder.builder()
            .password()
            .refreshToken()
            .build()
    val authorizedClientManager = DefaultOAuth2AuthorizedClientManager(
            clientRegistrationRepository, authorizedClientRepository)
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider)

    // Assuming the `username` and `password` are supplied as `HttpServletRequest`
parameters,
    // map the `HttpServletRequest` parameters to
`OAuth2AuthorizationContext.getAttributes()`
    authorizedClientManager.setContextAttributesMapper(contextAttributesMapper())
    return authorizedClientManager
}

private fun contextAttributesMapper(): Function<OAuth2AuthorizeRequest,
MutableMap<String, Any>> {
    return Function { authorizeRequest ->
        var contextAttributes: MutableMap<String, Any> = mutableMapOf()
        val servletRequest: HttpServletRequest =
authorizeRequest.getAttribute(HttpServletRequest::class.java.name)
        val username = servletRequest.getParameter(OAuth2ParameterNames.USERNAME)
        val password = servletRequest.getParameter(OAuth2ParameterNames.PASSWORD)
        if (StringUtils.hasText(username) && StringUtils.hasText(password)) {
            contextAttributes = hashMapOf()

            // `PasswordOAuth2AuthorizedClientProvider` requires both attributes
            contextAttributes[OAuth2AuthorizationContext.USERNAME_ATTRIBUTE_NAME]
= username
            contextAttributes[OAuth2AuthorizationContext.PASSWORD_ATTRIBUTE_NAME]
= password
        }
        contextAttributes
    }
}
```

You may obtain the `OAuth2AccessToken` as follows:

Java

```
@Controller
public class OAuth2ClientController {

    @Autowired
    private OAuth2AuthorizedClientManager authorizedClientManager;

    @GetMapping("/")
    public String index(Authentication authentication,
                        HttpServletRequest servletRequest,
                        HttpServletResponse servletResponse) {

        OAuth2AuthorizeRequest authorizeRequest =
OAuth2AuthorizeRequest.withClientRegistrationId("okta")
                .principal(authentication)
                .attributes(attrs -> {
                    attrs.put(HttpServletRequest.class.getName(), servletRequest);
                    attrs.put(HttpServletResponse.class.getName(),
servletResponse);
                })
                .build();
        OAuth2AuthorizedClient authorizedClient =
this.authorizedClientManager.authorize(authorizeRequest);

        OAuth2AccessToken accessToken = authorizedClient.getAccessToken();

        ...

        return "index";
    }
}
```

Kotlin

```
@Controller
class OAuth2ClientController {
    @Autowired
    private lateinit var authorizedClientManager: OAuth2AuthorizedClientManager

    @GetMapping("/")
    fun index(authentication: Authentication?,
              servletRequest: HttpServletRequest,
              servletResponse: HttpServletResponse): String {
        val authorizeRequest: OAuth2AuthorizeRequest =
OAuth2AuthorizeRequest.withClientRegistrationId("okta")
                .principal(authentication)
                .attributes(Consumer {
                    it[HttpServletRequest::class.java.name] = servletRequest
                    it[HttpServletResponse::class.java.name] = servletResponse
                })
                .build()
        val authorizedClient = authorizedClientManager.authorize(authorizeRequest)
        val accessToken: OAuth2AccessToken = authorizedClient.accessToken

        ...

        return "index"
    }
}
```

`HttpServletRequest` and `HttpServletResponse` are both OPTIONAL attributes. If not provided, it will default to `ServletRequestAttributes` using `RequestContextHolder.getRequestAttributes()`.

JWT Bearer

Please refer to JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants for further details on the JWT Bearer grant.

Requesting an Access Token

Please refer to the Access Token Request/Response protocol flow for the JWT Bearer grant.

The default implementation of `OAuth2AccessTokenResponseClient` for the JWT Bearer grant is `DefaultJwtBearerTokenResponseClient`, which uses a `RestOperations` when requesting an access token at the Authorization Server's Token Endpoint.

The `DefaultJwtBearerTokenResponseClient` is quite flexible as it allows you to customize the pre-processing of the Token Request and/or post-handling of the Token Response.

Customizing the Access Token Request

If you need to customize the pre-processing of the Token Request, you can provide `DefaultJwtBearerTokenResponseClient.setRequestEntityConverter()` with a custom `Converter<JwtBearerGrantRequest, RequestEntity<?>>`. The default implementation `JwtBearerGrantRequestEntityConverter` builds a `RequestEntity` representation of a OAuth 2.0 Access Token Request. However, providing a custom `Converter`, would allow you to extend the Token Request and add custom parameter(s).

Customizing the Access Token Response

On the other end, if you need to customize the post-handling of the Token Response, you will need to provide `DefaultJwtBearerTokenResponseClient.setRestOperations()` with a custom configured `RestOperations`. The default `RestOperations` is configured as follows:

Java

```
RestTemplate restTemplate = new RestTemplate(Arrays.asList(
        new FormHttpMessageConverter(),
        new OAuth2AccessTokenResponseHttpMessageConverter()));

restTemplate.setErrorHandler(new OAuth2ErrorResponseErrorHandler());
```

Kotlin

```
val restTemplate = RestTemplate(listOf(
        FormHttpMessageConverter(),
        OAuth2AccessTokenResponseHttpMessageConverter()))

restTemplate.errorHandler = OAuth2ErrorResponseErrorHandler()
```

Spring MVC `FormHttpMessageConverter` is required as it's used when sending the OAuth 2.0 Access Token Request.

`OAuth2AccessTokenResponseHttpMessageConverter` is a `HttpMessageConverter` for an OAuth 2.0 Access Token Response. You can provide `OAuth2AccessTokenResponseHttpMessageConverter.setTokenResponseConverter()` with a custom `Converter<Map<String, String>, OAuth2AccessTokenResponse>` that is used for converting the OAuth 2.0 Access Token Response parameters to an `OAuth2AccessTokenResponse`.

`OAuth2ErrorResponseErrorHandler` is a `ResponseErrorHandler` that can handle an OAuth 2.0 Error, eg. 400 Bad Request. It uses an `OAuth2ErrorHttpMessageConverter` for converting the OAuth 2.0 Error parameters to an `OAuth2Error`.

Whether you customize `DefaultJwtBearerTokenResponseClient` or provide your own implementation of `OAuth2AccessTokenResponseClient`, you'll need to configure it as shown in the following example:

Java

```
// Customize
OAuth2AccessTokenResponseClient<JwtBearerGrantRequest>
jwtBearerTokenResponseClient = ...

JwtBearerOAuth2AuthorizedClientProvider jwtBearerAuthorizedClientProvider = new
JwtBearerOAuth2AuthorizedClientProvider();
jwtBearerAuthorizedClientProvider.setAccessTokenResponseClient(jwtBearerTokenRespo
nseClient);

OAuth2AuthorizedClientProvider authorizedClientProvider =
        OAuth2AuthorizedClientProviderBuilder.builder()
                .provider(jwtBearerAuthorizedClientProvider)
                .build();

...

authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);
```

Kotlin

```
// Customize
val jwtBearerTokenResponseClient:
OAuth2AccessTokenResponseClient<JwtBearerGrantRequest> = ...

val jwtBearerAuthorizedClientProvider = JwtBearerOAuth2AuthorizedClientProvider()
jwtBearerAuthorizedClientProvider.setAccessTokenResponseClient(jwtBearerTokenRespo
nseClient);

val authorizedClientProvider = OAuth2AuthorizedClientProviderBuilder.builder()
        .provider(jwtBearerAuthorizedClientProvider)
        .build()

...

authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider)
```

Using the Access Token

Given the following Spring Boot 2.x properties for an OAuth 2.0 Client registration:

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: okta-client-id
            client-secret: okta-client-secret
            authorization-grant-type: urn:ietf:params:oauth:grant-type:jwt-bearer
            scope: read
        provider:
          okta:
            token-uri: https://dev-1234.oktapreview.com/oauth2/v1/token
```

...and the `OAuth2AuthorizedClientManager` `@Bean`:

Java

```
@Bean
public OAuth2AuthorizedClientManager authorizedClientManager(
        ClientRegistrationRepository clientRegistrationRepository,
        OAuth2AuthorizedClientRepository authorizedClientRepository) {

    JwtBearerOAuth2AuthorizedClientProvider jwtBearerAuthorizedClientProvider =
            new JwtBearerOAuth2AuthorizedClientProvider();

    OAuth2AuthorizedClientProvider authorizedClientProvider =
            OAuth2AuthorizedClientProviderBuilder.builder()
                    .provider(jwtBearerAuthorizedClientProvider)
                    .build();

    DefaultOAuth2AuthorizedClientManager authorizedClientManager =
            new DefaultOAuth2AuthorizedClientManager(
                    clientRegistrationRepository, authorizedClientRepository);
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);

    return authorizedClientManager;
}
```

Kotlin

```
@Bean
fun authorizedClientManager(
        clientRegistrationRepository: ClientRegistrationRepository,
        authorizedClientRepository: OAuth2AuthorizedClientRepository):
OAuth2AuthorizedClientManager {
    val jwtBearerAuthorizedClientProvider =
JwtBearerOAuth2AuthorizedClientProvider()
    val authorizedClientProvider = OAuth2AuthorizedClientProviderBuilder.builder()
            .provider(jwtBearerAuthorizedClientProvider)
            .build()
    val authorizedClientManager = DefaultOAuth2AuthorizedClientManager(
            clientRegistrationRepository, authorizedClientRepository)
    authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider)
    return authorizedClientManager
}
```

You may obtain the `OAuth2AccessToken` as follows:

Java

```
@RestController
public class OAuth2ResourceServerController {

    @Autowired
    private OAuth2AuthorizedClientManager authorizedClientManager;

    @GetMapping("/resource")
    public String resource(JwtAuthenticationToken jwtAuthentication) {
        OAuth2AuthorizeRequest authorizeRequest =
OAuth2AuthorizeRequest.withClientRegistrationId("okta")
                .principal(jwtAuthentication)
                .build();
        OAuth2AuthorizedClient authorizedClient =
this.authorizedClientManager.authorize(authorizeRequest);
        OAuth2AccessToken accessToken = authorizedClient.getAccessToken();

        ...

    }
}
```

Kotlin

```
class OAuth2ResourceServerController {

    @Autowired
    private lateinit var authorizedClientManager: OAuth2AuthorizedClientManager

    @GetMapping("/resource")
    fun resource(jwtAuthentication: JwtAuthenticationToken?): String {
        val authorizeRequest: OAuth2AuthorizeRequest =
OAuth2AuthorizeRequest.withClientRegistrationId("okta")
                .principal(jwtAuthentication)
                .build()
        val authorizedClient = authorizedClientManager.authorize(authorizeRequest)
        val accessToken: OAuth2AccessToken = authorizedClient.accessToken

        ...

    }
}
```

12.2.3. Client Authentication Support

JWT Bearer

Please refer to JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants for further details on JWT Bearer Client Authentication.

The default implementation for JWT Bearer Client Authentication is `NimbusJwtClientAuthenticationParametersConverter`, which is a `Converter` that customizes the Token Request parameters by adding a signed JSON Web Token (JWS) in the `client_assertion` parameter.

The `java.security.PrivateKey` or `javax.crypto.SecretKey` used for signing the JWS is supplied by the `com.nimbusds.jose.jwk.JWK` resolver associated with `NimbusJwtClientAuthenticationParametersConverter`.

Authenticate using `private_key_jwt`

Given the following Spring Boot 2.x properties for an OAuth 2.0 Client registration:

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: okta-client-id
            client-authentication-method: private_key_jwt
            authorization-grant-type: authorization_code
            ...
```

The following example shows how to configure `DefaultAuthorizationCodeTokenResponseClient`:

Java

```
Function<ClientRegistration, JWK> jwkResolver = (clientRegistration) -> {
    if
(clientRegistration.getClientAuthenticationMethod().equals(ClientAuthenticationMet
hod.PRIVATE_KEY_JWT)) {
        // Assuming RSA key type
        RSAPublicKey publicKey = ...
        RSAPrivateKey privateKey = ...
        return new RSAKey.Builder(publicKey)
                .privateKey(privateKey)
                .keyID(UUID.randomUUID().toString())
                .build();
    }
    return null;
};

OAuth2AuthorizationCodeGrantRequestEntityConverter requestEntityConverter =
        new OAuth2AuthorizationCodeGrantRequestEntityConverter();
requestEntityConverter.addParametersConverter(
        new NimbusJwtClientAuthenticationParametersConverter<>(jwkResolver));

DefaultAuthorizationCodeTokenResponseClient tokenResponseClient =
        new DefaultAuthorizationCodeTokenResponseClient();
tokenResponseClient.setRequestEntityConverter(requestEntityConverter);
```

Kotlin

```
val jwkResolver: Function<ClientRegistration, JWK> =
    Function<ClientRegistration, JWK> { clientRegistration ->
        if
(clientRegistration.clientAuthenticationMethod.equals(ClientAuthenticationMethod.P
RIVATE_KEY_JWT)) {
            // Assuming RSA key type
            var publicKey: RSAPublicKey
            var privateKey: RSAPrivateKey
            RSAKey.Builder(publicKey) = //...
                .privateKey(privateKey) = //...
                .keyID(UUID.randomUUID().toString())
                .build()
        }
        null
    }

val requestEntityConverter = OAuth2AuthorizationCodeGrantRequestEntityConverter()
requestEntityConverter.addParametersConverter(
    NimbusJwtClientAuthenticationParametersConverter(jwkResolver)
)

val tokenResponseClient = DefaultAuthorizationCodeTokenResponseClient()
tokenResponseClient.setRequestEntityConverter(requestEntityConverter)
```

Authenticate using `client_secret_jwt`

Given the following Spring Boot 2.x properties for an OAuth 2.0 Client registration:

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: okta-client-id
            client-secret: okta-client-secret
            client-authentication-method: client_secret_jwt
            authorization-grant-type: client_credentials
            ...
```

The following example shows how to configure `DefaultClientCredentialsTokenResponseClient`:

Java

```
Function<ClientRegistration, JWK> jwkResolver = (clientRegistration) -> {
    if
(clientRegistration.getClientAuthenticationMethod().equals(ClientAuthenticationMet
hod.CLIENT_SECRET_JWT)) {
        SecretKeySpec secretKey = new SecretKeySpec(

clientRegistration.getClientSecret().getBytes(StandardCharsets.UTF_8),
                "HmacSHA256");
        return new OctetSequenceKey.Builder(secretKey)
                .keyID(UUID.randomUUID().toString())
                .build();
    }
    return null;
};

OAuth2ClientCredentialsGrantRequestEntityConverter requestEntityConverter =
        new OAuth2ClientCredentialsGrantRequestEntityConverter();
requestEntityConverter.addParametersConverter(
        new NimbusJwtClientAuthenticationParametersConverter<>(jwkResolver));

DefaultClientCredentialsTokenResponseClient tokenResponseClient =
        new DefaultClientCredentialsTokenResponseClient();
tokenResponseClient.setRequestEntityConverter(requestEntityConverter);
```

Kotlin

```
val jwkResolver = Function<ClientRegistration, JWK?> { clientRegistration:
ClientRegistration ->
    if (clientRegistration.clientAuthenticationMethod ==
ClientAuthenticationMethod.CLIENT_SECRET_JWT) {
        val secretKey = SecretKeySpec(
            clientRegistration.clientSecret.toByteArray(StandardCharsets.UTF_8),
            "HmacSHA256"
        )
        OctetSequenceKey.Builder(secretKey)
            .keyID(UUID.randomUUID().toString())
            .build()
    }
    null
}

val requestEntityConverter = OAuth2ClientCredentialsGrantRequestEntityConverter()
requestEntityConverter.addParametersConverter(
    NimbusJwtClientAuthenticationParametersConverter(jwkResolver)
)

val tokenResponseClient = DefaultClientCredentialsTokenResponseClient()
tokenResponseClient.setRequestEntityConverter(requestEntityConverter)
```

12.2.4. Additional Features

Resolving an Authorized Client

The `@RegisteredOAuth2AuthorizedClient` annotation provides the capability of resolving a method parameter to an argument value of type `OAuth2AuthorizedClient`. This is a convenient alternative compared to accessing the `OAuth2AuthorizedClient` using the `OAuth2AuthorizedClientManager` or `OAuth2AuthorizedClientService`.

Java

```
@Controller
public class OAuth2ClientController {

    @GetMapping("/")
    public String index(@RegisteredOAuth2AuthorizedClient("okta")
OAuth2AuthorizedClient authorizedClient) {
        OAuth2AccessToken accessToken = authorizedClient.getAccessToken();

        ...

        return "index";
    }
}
```

Kotlin

```
@Controller
class OAuth2ClientController {
    @GetMapping("/")
    fun index(@RegisteredOAuth2AuthorizedClient("okta") authorizedClient:
OAuth2AuthorizedClient): String {
        val accessToken = authorizedClient.accessToken

        ...

        return "index"
    }
}
```

The `@RegisteredOAuth2AuthorizedClient` annotation is handled by `OAuth2AuthorizedClientArgumentResolver`, which directly uses an OAuth2AuthorizedClientManager and therefore inherits it's capabilities.

12.2.5. WebClient integration for Servlet Environments

The OAuth 2.0 Client support integrates with `WebClient` using an `ExchangeFilterFunction`.

The `ServletOAuth2AuthorizedClientExchangeFilterFunction` provides a simple mechanism for requesting protected resources by using an `OAuth2AuthorizedClient` and including the associated `OAuth2AccessToken` as a Bearer Token. It directly uses an OAuth2AuthorizedClientManager and therefore inherits the following capabilities:

- An `OAuth2AccessToken` will be requested if the client has not yet been authorized.
  - `authorization_code` - triggers the Authorization Request redirect to initiate the flow
  - `client_credentials` - the access token is obtained directly from the Token Endpoint

    - `password` - the access token is obtained directly from the Token Endpoint
- If the `OAuth2AccessToken` is expired, it will be refreshed (or renewed) if an `OAuth2AuthorizedClientProvider` is available to perform the authorization

The following code shows an example of how to configure `WebClient` with OAuth 2.0 Client support:

Java

```
@Bean
WebClient webClient(OAuth2AuthorizedClientManager authorizedClientManager) {
    ServletOAuth2AuthorizedClientExchangeFilterFunction oauth2Client =
            new
ServletOAuth2AuthorizedClientExchangeFilterFunction(authorizedClientManager);
    return WebClient.builder()
            .apply(oauth2Client.oauth2Configuration())
            .build();
}
```

Kotlin

```
@Bean
fun webClient(authorizedClientManager: OAuth2AuthorizedClientManager?): WebClient
{
    val oauth2Client =
ServletOAuth2AuthorizedClientExchangeFilterFunction(authorizedClientManager)
    return WebClient.builder()
            .apply(oauth2Client.oauth2Configuration())
            .build()
}
```

Providing the Authorized Client

The `ServletOAuth2AuthorizedClientExchangeFilterFunction` determines the client to use (for a request) by resolving the `OAuth2AuthorizedClient` from the `ClientRequest.attributes()` (request attributes).

The following code shows how to set an `OAuth2AuthorizedClient` as a request attribute:

Java

```
@GetMapping("/")
public String index(@RegisteredOAuth2AuthorizedClient("okta")
OAuth2AuthorizedClient authorizedClient) {
    String resourceUri = ...

    String body = webClient
            .get()
            .uri(resourceUri)
            .attributes(oauth2AuthorizedClient(authorizedClient))   ①
            .retrieve()
            .bodyToMono(String.class)
            .block();

    ...

    return "index";
}
```

Kotlin

```
@GetMapping("/")
fun index(@RegisteredOAuth2AuthorizedClient("okta") authorizedClient:
OAuth2AuthorizedClient): String {
    val resourceUri: String = ...
    val body: String = webClient
            .get()
            .uri(resourceUri)
            .attributes(oauth2AuthorizedClient(authorizedClient)) ①
            .retrieve()
            .bodyToMono()
            .block()

    ...

    return "index"
}
```

① `oauth2AuthorizedClient()` is a `static` method in `ServletOAuth2AuthorizedClientExchangeFilterFunction`.

The following code shows how to set the `ClientRegistration.getRegistrationId()` as a request attribute:

Java

```
@GetMapping("/")
public String index() {
    String resourceUri = ...

    String body = webClient
            .get()
            .uri(resourceUri)
            .attributes(clientRegistrationId("okta"))   ①
            .retrieve()
            .bodyToMono(String.class)
            .block();

    ...

    return "index";
}
```

Kotlin

```
@GetMapping("/")
fun index(): String {
    val resourceUri: String = ...

    val body: String = webClient
            .get()
            .uri(resourceUri)
            .attributes(clientRegistrationId("okta"))  ①
            .retrieve()
            .bodyToMono()
            .block()

    ...

    return "index"
}
```

① `clientRegistrationId()` is a `static` method in `ServletOAuth2AuthorizedClientExchangeFilterFunction`.

Defaulting the Authorized Client

If neither `OAuth2AuthorizedClient` or `ClientRegistration.getRegistrationId()` is provided as a request attribute, the `ServletOAuth2AuthorizedClientExchangeFilterFunction` can determine the *default* client to use depending on it's configuration.

If `setDefaultOAuth2AuthorizedClient(true)` is configured and the user has authenticated using

`HttpSecurity.oauth2Login()`, the `OAuth2AccessToken` associated with the current `OAuth2AuthenticationToken` is used.

The following code shows the specific configuration:

Java

```
@Bean
WebClient webClient(OAuth2AuthorizedClientManager authorizedClientManager) {
    ServletOAuth2AuthorizedClientExchangeFilterFunction oauth2Client =
            new
ServletOAuth2AuthorizedClientExchangeFilterFunction(authorizedClientManager);
    oauth2Client.setDefaultOAuth2AuthorizedClient(true);
    return WebClient.builder()
            .apply(oauth2Client.oauth2Configuration())
            .build();
}
```

Kotlin

```
@Bean
fun webClient(authorizedClientManager: OAuth2AuthorizedClientManager?): WebClient
{
    val oauth2Client =
ServletOAuth2AuthorizedClientExchangeFilterFunction(authorizedClientManager)
    oauth2Client.setDefaultOAuth2AuthorizedClient(true)
    return WebClient.builder()
            .apply(oauth2Client.oauth2Configuration())
            .build()
}
```

It is recommended to be cautious with this feature since all HTTP requests will receive the access token.

Alternatively, if `setDefaultClientRegistrationId("okta")` is configured with a valid `ClientRegistration`, the `OAuth2AccessToken` associated with the `OAuth2AuthorizedClient` is used.

The following code shows the specific configuration:

Java

```
@Bean
WebClient webClient(OAuth2AuthorizedClientManager authorizedClientManager) {
    ServletOAuth2AuthorizedClientExchangeFilterFunction oauth2Client =
            new
ServletOAuth2AuthorizedClientExchangeFilterFunction(authorizedClientManager);
    oauth2Client.setDefaultClientRegistrationId("okta");
    return WebClient.builder()
            .apply(oauth2Client.oauth2Configuration())
            .build();
}
```

Kotlin

```
@Bean
fun webClient(authorizedClientManager: OAuth2AuthorizedClientManager?): WebClient
{
    val oauth2Client =
ServletOAuth2AuthorizedClientExchangeFilterFunction(authorizedClientManager)
    oauth2Client.setDefaultClientRegistrationId("okta")
    return WebClient.builder()
            .apply(oauth2Client.oauth2Configuration())
            .build()
}
```

It is recommended to be cautious with this feature since all HTTP requests will receive the access token.

12.3. OAuth 2.0 Resource Server

Spring Security supports protecting endpoints using two forms of OAuth 2.0 Bearer Tokens:

- JWT
- Opaque Tokens

This is handy in circumstances where an application has delegated its authority management to an authorization server (for example, Okta or Ping Identity). This authorization server can be consulted by resource servers to authorize requests.

This section provides details on how Spring Security provides support for OAuth 2.0 Bearer Tokens.

Working samples for both JWTs and Opaque Tokens are available in the Spring Security Samples repository.

Let's take a look at how Bearer Token Authentication works within Spring Security. First, we see

that, like Basic Authentication, the WWW-Authenticate header is sent back to an unauthenticated client.

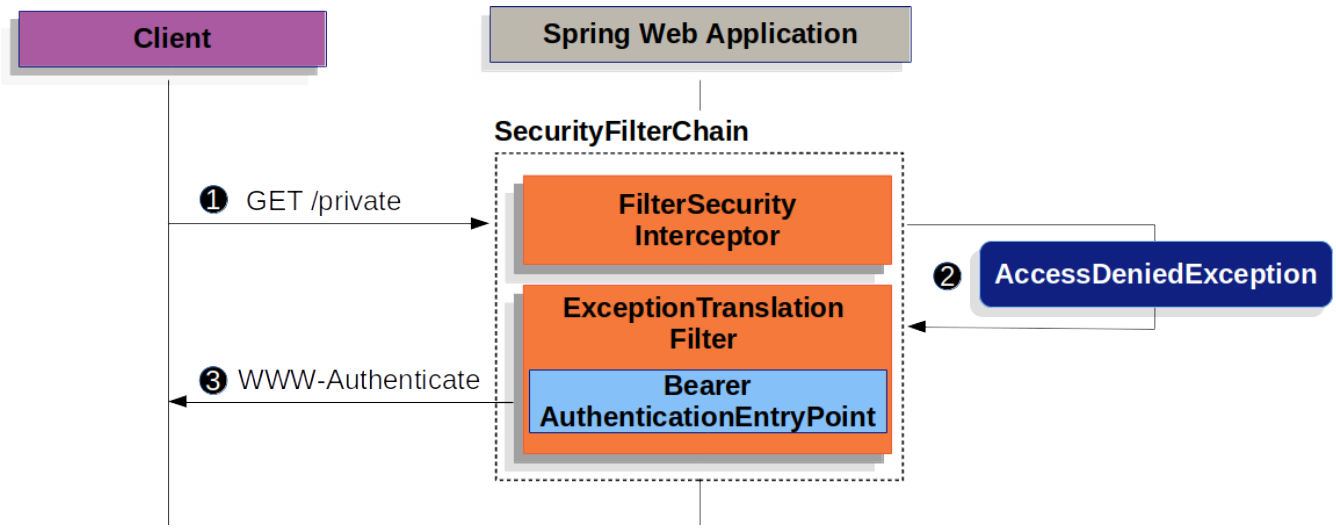


Figure 14. Sending WWW-Authenticate Header

The figure above builds off our `SecurityFilterChain` diagram.

1 First, a user makes an unauthenticated request to the resource `/private` for which it is not authorized.

2 Spring Security's `FilterSecurityInterceptor` indicates that the unauthenticated request is *Denied* by throwing an `AccessDeniedException`.

3 Since the user is not authenticated, `ExceptionTranslationFilter` initiates *Start Authentication*. The configured `AuthenticationEntryPoint` is an instance of `BearerTokenAuthenticationEntryPoint` which sends a WWW-Authenticate header. The `RequestCache` is typically a `NullRequestCache` that does not save the request since the client is capable of replaying the requests it originally requested.

When a client receives the `WWW-Authenticate: Bearer` header, it knows it should retry with a bearer token. Below is the flow for the bearer token being processed.

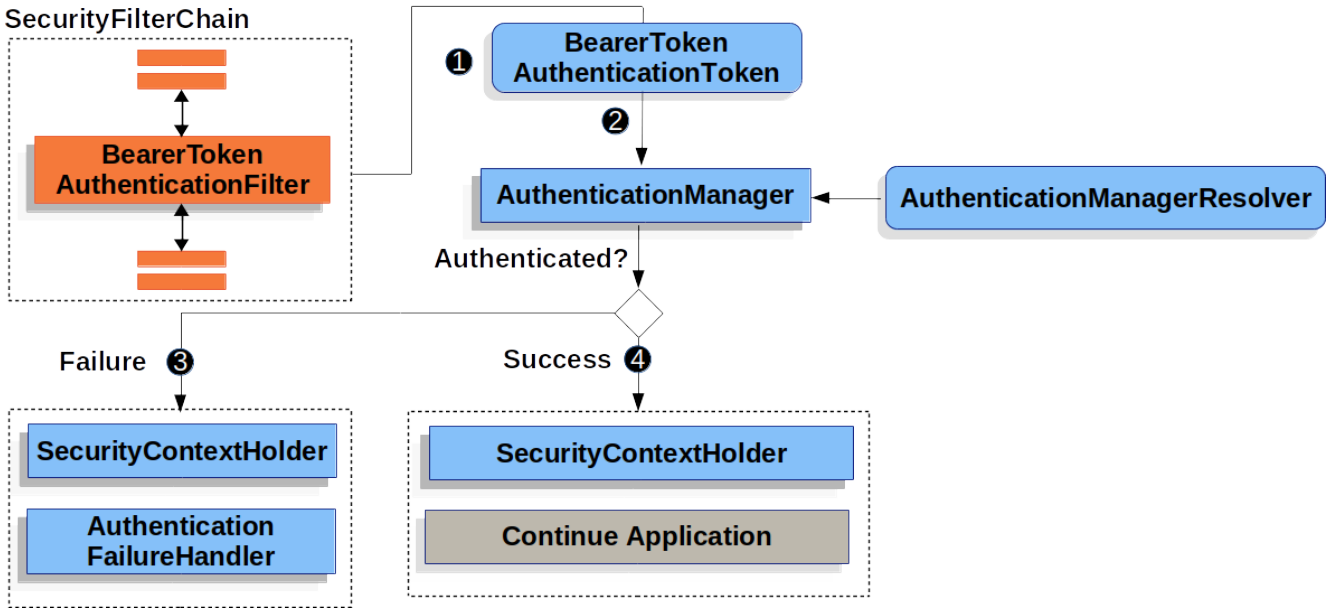


Figure 15. Authenticating Bearer Token

The figure builds off our `SecurityFilterChain` diagram.

1. When the user submits their bearer token, the `BearerTokenAuthenticationFilter` creates a `BearerTokenAuthenticationToken` which is a type of `Authentication` by extracting the token from the `HttpServletRequest`.

2. Next, the `HttpServletRequest` is passed to the `AuthenticationManagerResolver`, which selects the `AuthenticationManager`. The `BearerTokenAuthenticationToken` is passed into the `AuthenticationManager` to be authenticated. The details of what `AuthenticationManager` looks like depends on whether you're configured for JWT or opaque token.

3. If authentication fails, then *Failure*

- The SecurityContextHolder is cleared out.
- The `AuthenticationEntryPoint` is invoked to trigger the WWW-Authenticate header to be sent again.

4. If authentication is successful, then *Success*.

- The Authentication is set on the SecurityContextHolder.
- The `BearerTokenAuthenticationFilter` invokes `FilterChain.doFilter(request,response)` to continue with the rest of the application logic.

12.3.1. Minimal Dependencies for JWT

Most Resource Server support is collected into `spring-security-oauth2-resource-server`. However, the support for decoding and verifying JWTs is in `spring-security-oauth2-jose`, meaning that both are necessary in order to have a working resource server that supports JWT-encoded Bearer Tokens.

12.3.2. Minimal Configuration for JWTs

When using Spring Boot, configuring an application as a resource server consists of two basic steps. First, include the needed dependencies and second, indicate the location of the authorization server.

Specifying the Authorization Server

In a Spring Boot application, to specify which authorization server to use, simply do:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://idp.example.com/issuer
```

Where `https://idp.example.com/issuer` is the value contained in the `iss` claim for JWT tokens that

the authorization server will issue. Resource Server will use this property to further self-configure, discover the authorization server's public keys, and subsequently validate incoming JWTs.

To use the `issuer-uri` property, it must also be true that one of `https://idp.example.com/issuer/.well-known/openid-configuration`, `https://idp.example.com/.well-known/openid-configuration/issuer`, or `https://idp.example.com/.well-known/oauth-authorization-server/issuer` is a supported endpoint for the authorization server. This endpoint is referred to as a Provider Configuration endpoint or a Authorization Server Metadata endpoint.

And that's it!

Startup Expectations

When this property and these dependencies are used, Resource Server will automatically configure itself to validate JWT-encoded Bearer Tokens.

It achieves this through a deterministic startup process:

1. Query the Provider Configuration or Authorization Server Metadata endpoint for the `jwks_url` property
2. Query the `jwks_url` endpoint for supported algorithms
3. Configure the validation strategy to query `jwks_url` for valid public keys of the algorithms found
4. Configure the validation strategy to validate each JWTs `iss` claim against `https://idp.example.com`.

A consequence of this process is that the authorization server must be up and receiving requests in order for Resource Server to successfully start up.

If the authorization server is down when Resource Server queries it (given appropriate timeouts), then startup will fail.

Runtime Expectations

Once the application is started up, Resource Server will attempt to process any request containing an `Authorization: Bearer` header:

```
GET / HTTP/1.1
Authorization: Bearer some-token-value # Resource Server will process this
```

So long as this scheme is indicated, Resource Server will attempt to process the request according to the Bearer Token specification.

Given a well-formed JWT, Resource Server will:

1. Validate its signature against a public key obtained from the `jwks_url` endpoint during startup and matched against the JWT

2. Validate the JWT's `exp` and `nbf` timestamps and the JWT's `iss` claim, and
3. Map each scope to an authority with the prefix `SCOPE_`.

As the authorization server makes available new keys, Spring Security will automatically rotate the keys used to validate JWTs.

The resulting `Authentication#getPrincipal`, by default, is a Spring Security `Jwt` object, and `Authentication#getName` maps to the JWT's `sub` property, if one is present.

From here, consider jumping to:

- How JWT Authentication Works
- How to Configure without tying Resource Server startup to an authorization server's availability
- How to Configure without Spring Boot

12.3.3. How JWT Authentication Works

Next, let's see the architectural components that Spring Security uses to support JWT Authentication in servlet-based applications, like the one we just saw.

`JwtAuthenticationProvider` is an `AuthenticationProvider` implementation that leverages a `JwtDecoder` and `JwtAuthenticationConverter` to authenticate a JWT.

Let's take a look at how `JwtAuthenticationProvider` works within Spring Security. The figure explains details of how the `AuthenticationManager` in figures from Reading the Bearer Token works.

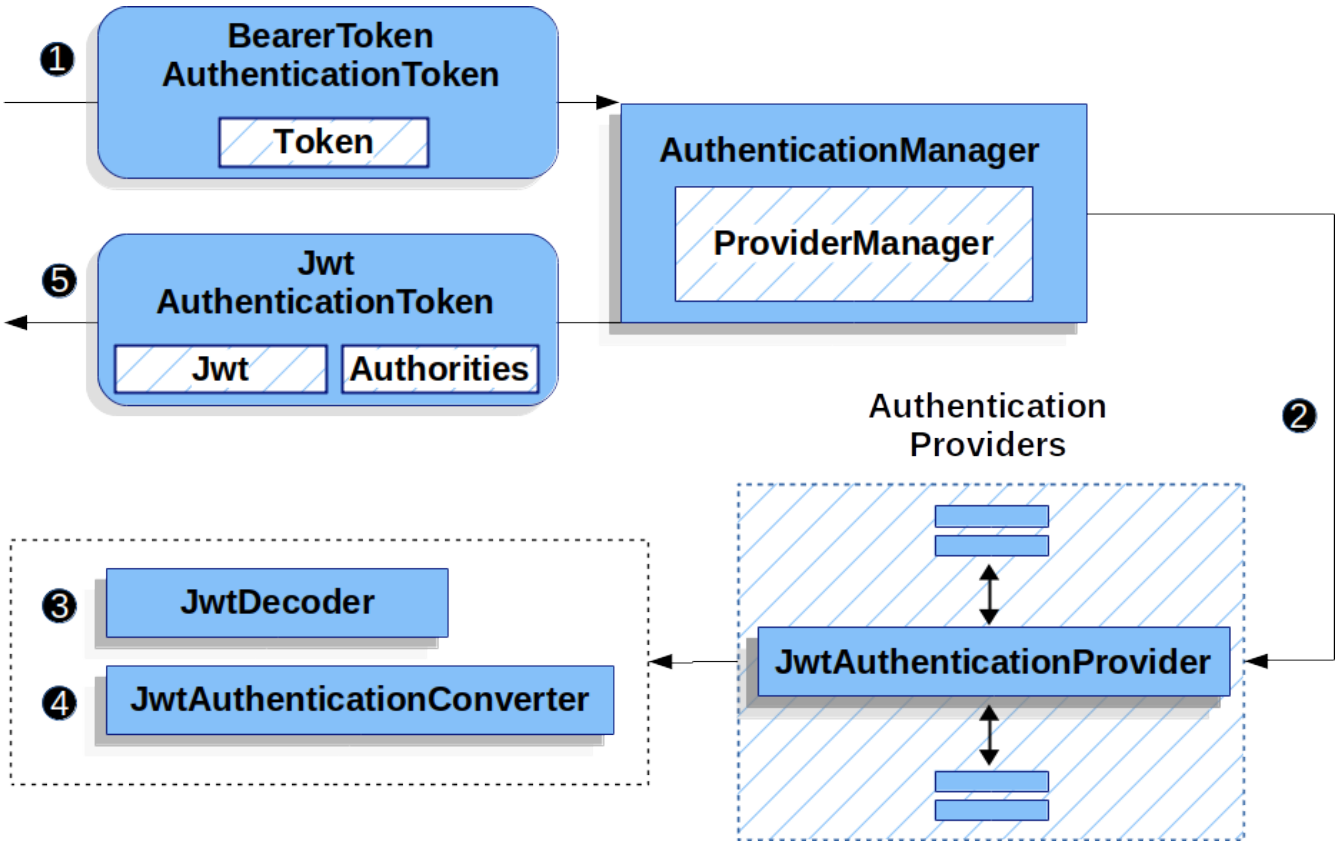


Figure 16. `JwtAuthenticationProvider` Usage

1. The authentication `Filter` from Reading the Bearer Token passes a `BearerTokenAuthenticationToken` to the `AuthenticationManager` which is implemented by `ProviderManager`.

2. The `ProviderManager` is configured to use an AuthenticationProvider of type `JwtAuthenticationProvider`.

3. `JwtAuthenticationProvider` decodes, verifies, and validates the `Jwt` using a `JwtDecoder`.

4. `JwtAuthenticationProvider` then uses the `JwtAuthenticationConverter` to convert the `Jwt` into a `Collection` of granted authorities.

5. When authentication is successful, the `Authentication` that is returned is of type `JwtAuthenticationToken` and has a principal that is the `Jwt` returned by the configured `JwtDecoder`. Ultimately, the returned `JwtAuthenticationToken` will be set on the `SecurityContextHolder` by the authentication `Filter`.

12.3.4. Specifying the Authorization Server JWK Set Uri Directly

If the authorization server doesn't support any configuration endpoints, or if Resource Server must be able to start up independently from the authorization server, then the `jwk-set-uri` can be supplied as well:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://idp.example.com
          jwk-set-uri: https://idp.example.com/.well-known/jwks.json
```

The JWK Set uri is not standardized, but can typically be found in the authorization server's documentation

Consequently, Resource Server will not ping the authorization server at startup. We still specify the `issuer-uri` so that Resource Server still validates the `iss` claim on incoming JWTs.

This property can also be supplied directly on the DSL.

12.3.5. Overriding or Replacing Boot Auto Configuration

There are two `@Bean` s that Spring Boot generates on Resource Server's behalf.

The first is a `WebSecurityConfigurerAdapter` that configures the app as a resource server. When including `spring-security-oauth2-jose`, this `WebSecurityConfigurerAdapter` looks like:

Example 120. Default JWT Configuration

Java

```
protected void configure(HttpSecurity http) {
    http
        .authorizeRequests(authorize -> authorize
            .anyRequest().authenticated()
        )
        .oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt);
}
```

Kotlin

```
fun configure(http: HttpSecurity) {
    http {
        authorizeRequests {
            authorize(anyRequest, authenticated)
        }
        oauth2ResourceServer {
            jwt { }
        }
    }
}
```

If the application doesn't expose a `WebSecurityConfigurerAdapter` bean, then Spring Boot will expose the above default one.

Replacing this is as simple as exposing the bean within the application:

Example 121. Custom JWT Configuration

Java

```
@EnableWebSecurity
public class MyCustomSecurityConfiguration extends WebSecurityConfigurerAdapter {
    protected void configure(HttpSecurity http) {
        http
            .authorizeRequests(authorize -> authorize
                .mvcMatchers("/messages/**").hasAuthority("SCOPE_message:read")
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt
                    .jwtAuthenticationConverter(myConverter())
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class MyCustomSecurityConfiguration : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            authorizeRequests {
                authorize("/messages/**", hasAuthority("SCOPE_message:read"))
                authorize(anyRequest, authenticated)
            }
            oauth2ResourceServer {
                jwt {
                    jwtAuthenticationConverter = myConverter()
                }
            }
        }
    }
}
```

The above requires the scope of `message:read` for any URL that starts with `/messages/`.

Methods on the `oauth2ResourceServer` DSL will also override or replace auto configuration.

For example, the second `@Bean` Spring Boot creates is a `JwtDecoder`, which decodes `String` tokens into validated instances of `Jwt`:

Example 122. JWT Decoder

Java

```
@Bean
public JwtDecoder jwtDecoder() {
    return JwtDecoders.fromIssuerLocation(issuerUri);
}
```

Kotlin

```
@Bean
fun jwtDecoder(): JwtDecoder {
    return JwtDecoders.fromIssuerLocation(issuerUri)
}
```

Calling `JwtDecoders#fromIssuerLocation` is what invokes the Provider Configuration or Authorization Server Metadata endpoint in order to derive the JWK Set Uri.

If the application doesn't expose a `JwtDecoder` bean, then Spring Boot will expose the above default one.

And its configuration can be overridden using `jwkSetUri()` or replaced using `decoder()`.

Or, if you're not using Spring Boot at all, then both of these components - the filter chain and a `JwtDecoder` can be specified in XML.

The filter chain is specified like so:

Example 123. Default JWT Configuration

Xml

```
<http>
    <intercept-uri pattern="/**" access="authenticated"/>
    <oauth2-resource-server>
        <jwt decoder-ref="jwtDecoder"/>
    </oauth2-resource-server>
</http>
```

And the `JwtDecoder` like so:

Example 124. JWT Decoder

Xml

```
<bean id="jwtDecoder"
        class="org.springframework.security.oauth2.jwt.JwtDecoders"
        factory-method="fromIssuerLocation">
    <constructor-arg value="${spring.security.oauth2.resourceserver.jwt.jwk-set-
uri}"/>
</bean>
```

Using `jwkSetUri()`

An authorization server's JWK Set Uri can be configured as a configuration property or it can be supplied in the DSL:

Example 125. JWK Set Uri Configuration

Java

```
@EnableWebSecurity
public class DirectlyConfiguredJwkSetUri extends WebSecurityConfigurerAdapter {
    protected void configure(HttpSecurity http) {
        http
            .authorizeRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt
                    .jwkSetUri("https://idp.example.com/.well-known/jwks.json")
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class DirectlyConfiguredJwkSetUri : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            authorizeRequests {
                authorize(anyRequest, authenticated)
            }
            oauth2ResourceServer {
                jwt {
                    jwkSetUri = "https://idp.example.com/.well-known/jwks.json"
                }
            }
        }
    }
}
```

Xml

```
<http>
    <intercept-uri pattern="/**" access="authenticated"/>
    <oauth2-resource-server>
        <jwt jwk-set-uri="https://idp.example.com/.well-known/jwks.json"/>
    </oauth2-resource-server>
</http>
```

Using `jwkSetUri()` takes precedence over any configuration property.

Using `decoder()`

More powerful than `jwkSetUri()` is `decoder()`, which will completely replace any Boot auto configuration of `JwtDecoder`:

Example 126. JWT Decoder Configuration

Java

```
@EnableWebSecurity
public class DirectlyConfiguredJwtDecoder extends WebSecurityConfigurerAdapter {
    protected void configure(HttpSecurity http) {
        http
            .authorizeRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt
                    .decoder(myCustomDecoder())
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class DirectlyConfiguredJwtDecoder : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            authorizeRequests {
                authorize(anyRequest, authenticated)
            }
            oauth2ResourceServer {
                jwt {
                    jwtDecoder = myCustomDecoder()
                }
            }
        }
    }
}
```

Xml

```
<http>
    <intercept-uri pattern="/**" access="authenticated"/>
    <oauth2-resource-server>
        <jwt decoder-ref="myCustomDecoder"/>
    </oauth2-resource-server>
</http>
```

This is handy when deeper configuration, like validation, mapping, or request timeouts, is necessary.

Exposing a `JwtDecoder` `@Bean`

Or, exposing a `JwtDecoder` `@Bean` has the same effect as `decoder()`:

Java

```
@Bean
public JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withJwkSetUri(jwkSetUri).build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(): JwtDecoder {
    return NimbusJwtDecoder.withJwkSetUri(jwkSetUri).build()
}
```

12.3.6. Configuring Trusted Algorithms

By default, `NimbusJwtDecoder`, and hence Resource Server, will only trust and verify tokens using `RS256`.

You can customize this via Spring Boot, the NimbusJwtDecoder builder, or from the JWK Set response.

Via Spring Boot

The simplest way to set the algorithm is as a property:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jws-algorithm: RS512
          jwk-set-uri: https://idp.example.org/.well-known/jwks.json
```

Using a Builder

For greater power, though, we can use a builder that ships with `NimbusJwtDecoder`:

Java

```
@Bean
JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withJwkSetUri(this.jwkSetUri)
            .jwsAlgorithm(RS512).build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(): JwtDecoder {
    return NimbusJwtDecoder.withJwkSetUri(this.jwkSetUri)
            .jwsAlgorithm(RS512).build()
}
```

Calling `jwsAlgorithm` more than once will configure `NimbusJwtDecoder` to trust more than one algorithm, like so:

Java

```
@Bean
JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withJwkSetUri(this.jwkSetUri)
            .jwsAlgorithm(RS512).jwsAlgorithm(ES512).build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(): JwtDecoder {
    return NimbusJwtDecoder.withJwkSetUri(this.jwkSetUri)
            .jwsAlgorithm(RS512).jwsAlgorithm(ES512).build()
}
```

Or, you can call `jwsAlgorithms`:

Java

```
@Bean
JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withJwkSetUri(this.jwkSetUri)
            .jwsAlgorithms(algorithms -> {
                    algorithms.add(RS512);
                    algorithms.add(ES512);
            }).build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(): JwtDecoder {
    return NimbusJwtDecoder.withJwkSetUri(this.jwkSetUri)
            .jwsAlgorithms {
                it.add(RS512)
                it.add(ES512)
            }.build()
}
```

From JWK Set response

Since Spring Security's JWT support is based off of Nimbus, you can use all it's great features as well.

For example, Nimbus has a `JWSKeySelector` implementation that will select the set of algorithms based on the JWK Set URI response. You can use it to generate a `NimbusJwtDecoder` like so:

Java

```
@Bean
public JwtDecoder jwtDecoder() {
    // makes a request to the JWK Set endpoint
    JWSKeySelector<SecurityContext> jwsKeySelector =
            JWSAlgorithmFamilyJWSKeySelector.fromJWKSetURL(this.jwkSetUrl);

    DefaultJWTProcessor<SecurityContext> jwtProcessor =
            new DefaultJWTProcessor<>();
    jwtProcessor.setJWSKeySelector(jwsKeySelector);

    return new NimbusJwtDecoder(jwtProcessor);
}
```

Kotlin

```
@Bean
fun jwtDecoder(): JwtDecoder {
    // makes a request to the JWK Set endpoint
    val jwsKeySelector: JWSKeySelector<SecurityContext> =
JWSAlgorithmFamilyJWSKeySelector.fromJWKSetURL<SecurityContext>(this.jwkSetUrl)
    val jwtProcessor: DefaultJWTProcessor<SecurityContext> = DefaultJWTProcessor()
    jwtProcessor.jwsKeySelector = jwsKeySelector
    return NimbusJwtDecoder(jwtProcessor)
}
```

12.3.7. Trusting a Single Asymmetric Key

Simpler than backing a Resource Server with a JWK Set endpoint is to hard-code an RSA public key. The public key can be provided via Spring Boot or by Using a Builder.

Via Spring Boot

Specifying a key via Spring Boot is quite simple. The key's location can be specified like so:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          public-key-location: classpath:my-key.pub
```

Or, to allow for a more sophisticated lookup, you can post-process the `RsaKeyConversionServicePostProcessor`:

Java

```
@Bean
BeanFactoryPostProcessor conversionServiceCustomizer() {
    return beanFactory ->
        beanFactory.getBean(RsaKeyConversionServicePostProcessor.class)
                .setResourceLoader(new CustomResourceLoader());
}
```

Kotlin

```
@Bean
fun conversionServiceCustomizer(): BeanFactoryPostProcessor {
    return BeanFactoryPostProcessor { beanFactory ->
        beanFactory.getBean<RsaKeyConversionServicePostProcessor>()
                .setResourceLoader(CustomResourceLoader())
    }
}
```

Specify your key's location:

```
key.location: hfds://my-key.pub
```

And then autowire the value:

Java

```
@Value("${key.location}")
RSAPublicKey key;
```

Kotlin

```
@Value("\${key.location}")
val key: RSAPublicKey? = null
```

Using a Builder

To wire an `RSAPublicKey` directly, you can simply use the appropriate `NimbusJwtDecoder` builder, like so:

Java

```
@Bean
public JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withPublicKey(this.key).build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(): JwtDecoder {
    return NimbusJwtDecoder.withPublicKey(this.key).build()
}
```

12.3.8. Trusting a Single Symmetric Key

Using a single symmetric key is also simple. You can simply load in your `SecretKey` and use the appropriate `NimbusJwtDecoder` builder, like so:

Java

```
@Bean
public JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withSecretKey(this.key).build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(): JwtDecoder {
    return NimbusJwtDecoder.withSecretKey(key).build()
}
```

12.3.9. Configuring Authorization

A JWT that is issued from an OAuth 2.0 Authorization Server will typically either have a `scope` or `scp` attribute, indicating the scopes (or authorities) it's been granted, for example:

`{ …, "scope" : "messages contacts"}`

When this is the case, Resource Server will attempt to coerce these scopes into a list of granted authorities, prefixing each scope with the string "SCOPE_".

This means that to protect an endpoint or method with a scope derived from a JWT, the corresponding expressions should include this prefix:

Example 127. Authorization Configuration

Java

```
@EnableWebSecurity
public class DirectlyConfiguredJwkSetUri extends WebSecurityConfigurerAdapter {
    protected void configure(HttpSecurity http) {
        http
            .authorizeRequests(authorize -> authorize
                .mvcMatchers("/contacts/**").hasAuthority("SCOPE_contacts")
                .mvcMatchers("/messages/**").hasAuthority("SCOPE_messages")
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt);
    }
}
```

Kotlin

```
@EnableWebSecurity
class DirectlyConfiguredJwkSetUri : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            authorizeRequests {
                authorize("/contacts/**", hasAuthority("SCOPE_contacts"))
                authorize("/messages/**", hasAuthority("SCOPE_messages"))
                authorize(anyRequest, authenticated)
            }
            oauth2ResourceServer {
                jwt { }
            }
        }
    }
}
```

Xml

```
<http>
    <intercept-uri pattern="/contacts/**"
access="hasAuthority('SCOPE_contacts')"/>
    <intercept-uri pattern="/messages/**"
access="hasAuthority('SCOPE_messages')"/>
    <oauth2-resource-server>
        <jwt jwk-set-uri="https://idp.example.org/.well-known/jwks.json"/>
    </oauth2-resource-server>
</http>
```

Or similarly with method security:

Java

```
@PreAuthorize("hasAuthority('SCOPE_messages')")
public List<Message> getMessages(...) {}
```

Kotlin

```
@PreAuthorize("hasAuthority('SCOPE_messages')")
fun getMessages(): List<Message> { }
```

Extracting Authorities Manually

However, there are a number of circumstances where this default is insufficient. For example, some authorization servers don't use the `scope` attribute, but instead have their own custom attribute. Or, at other times, the resource server may need to adapt the attribute or a composition of attributes into internalized authorities.

To this end, Spring Security ships with `JwtAuthenticationConverter`, which is responsible for converting a `Jwt` into an `Authentication`. By default, Spring Security will wire the `JwtAuthenticationProvider` with a default instance of `JwtAuthenticationConverter`.

As part of configuring a `JwtAuthenticationConverter`, you can supply a subsidiary converter to go from `Jwt` to a `Collection` of granted authorities.

Let's say that that your authorization server communicates authorities in a custom claim called `authorities`. In that case, you can configure the claim that `JwtAuthenticationConverter` should inspect, like so:

Example 128. Authorities Claim Configuration

Java

```
@Bean
public JwtAuthenticationConverter jwtAuthenticationConverter() {
    JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new
JwtGrantedAuthoritiesConverter();
    grantedAuthoritiesConverter.setAuthoritiesClaimName("authorities");

    JwtAuthenticationConverter jwtAuthenticationConverter = new
JwtAuthenticationConverter();

jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesCon
verter);
    return jwtAuthenticationConverter;
}
```

Kotlin

```
@Bean
fun jwtAuthenticationConverter(): JwtAuthenticationConverter {
    val grantedAuthoritiesConverter = JwtGrantedAuthoritiesConverter()
    grantedAuthoritiesConverter.setAuthoritiesClaimName("authorities")

    val jwtAuthenticationConverter = JwtAuthenticationConverter()

jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesCon
verter)
    return jwtAuthenticationConverter
}
```

Xml

```
<http>
    <intercept-uri pattern="/contacts/**"
access="hasAuthority('SCOPE_contacts')"/>
    <intercept-uri pattern="/messages/**"
access="hasAuthority('SCOPE_messages')"/>
    <oauth2-resource-server>
        <jwt jwk-set-uri="https://idp.example.org/.well-known/jwks.json"
                jwt-authentication-converter-ref="jwtAuthenticationConverter"/>
    </oauth2-resource-server>
</http>

<bean id="jwtAuthenticationConverter"

class="org.springframework.security.oauth2.server.resource.authentication.JwtAuthe
nticationConverter">
    <property name="jwtGrantedAuthoritiesConverter"
ref="jwtGrantedAuthoritiesConverter"/>
</bean>

<bean id="jwtGrantedAuthoritiesConverter"

class="org.springframework.security.oauth2.server.resource.authentication.JwtGrant
edAuthoritiesConverter">
    <property name="authoritiesClaimName" value="authorities"/>
</bean>
```

You can also configure the authority prefix to be different as well. Instead of prefixing each authority with `SCOPE_`, you can change it to `ROLE_` like so:

Example 129. Authorities Prefix Configuration

Java

```
@Bean
public JwtAuthenticationConverter jwtAuthenticationConverter() {
    JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new
JwtGrantedAuthoritiesConverter();
    grantedAuthoritiesConverter.setAuthorityPrefix("ROLE_");

    JwtAuthenticationConverter jwtAuthenticationConverter = new
JwtAuthenticationConverter();

jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesCon
verter);
    return jwtAuthenticationConverter;
}
```

Kotlin

```
@Bean
fun jwtAuthenticationConverter(): JwtAuthenticationConverter {
    val grantedAuthoritiesConverter = JwtGrantedAuthoritiesConverter()
    grantedAuthoritiesConverter.setAuthorityPrefix("ROLE_")

    val jwtAuthenticationConverter = JwtAuthenticationConverter()

jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesCon
verter)
    return jwtAuthenticationConverter
}
```

Xml

```
<http>
    <intercept-uri pattern="/contacts/**"
access="hasAuthority('SCOPE_contacts')"/>
    <intercept-uri pattern="/messages/**"
access="hasAuthority('SCOPE_messages')"/>
    <oauth2-resource-server>
        <jwt jwk-set-uri="https://idp.example.org/.well-known/jwks.json"
                jwt-authentication-converter-ref="jwtAuthenticationConverter"/>
    </oauth2-resource-server>
</http>

<bean id="jwtAuthenticationConverter"

class="org.springframework.security.oauth2.server.resource.authentication.JwtAuthe
nticationConverter">
    <property name="jwtGrantedAuthoritiesConverter"
ref="jwtGrantedAuthoritiesConverter"/>
</bean>

<bean id="jwtGrantedAuthoritiesConverter"

class="org.springframework.security.oauth2.server.resource.authentication.JwtGrant
edAuthoritiesConverter">
    <property name="authorityPrefix" value="ROLE_"/>
</bean>
```

Or, you can remove the prefix altogether by calling `JwtGrantedAuthoritiesConverter#setAuthorityPrefix("")`.

For more flexibility, the DSL supports entirely replacing the converter with any class that implements `Converter<Jwt, AbstractAuthenticationToken>`:

Java

```
static class CustomAuthenticationConverter implements Converter<Jwt,
AbstractAuthenticationToken> {
    public AbstractAuthenticationToken convert(Jwt jwt) {
        return new CustomAuthenticationToken(jwt);
    }
}

// ...

@EnableWebSecurity
public class CustomAuthenticationConverterConfig extends
WebSecurityConfigurerAdapter {
    protected void configure(HttpSecurity http) {
        http
            .authorizeRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt
                    .jwtAuthenticationConverter(new
CustomAuthenticationConverter())
                )
            );
    }
}
```

Kotlin

```
internal class CustomAuthenticationConverter : Converter<Jwt,
AbstractAuthenticationToken> {
    override fun convert(jwt: Jwt): AbstractAuthenticationToken {
        return CustomAuthenticationToken(jwt)
    }
}

// ...

@EnableWebSecurity
class CustomAuthenticationConverterConfig : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
       http {
            authorizeRequests {
                authorize(anyRequest, authenticated)
            }
           oauth2ResourceServer {
               jwt {
                   jwtAuthenticationConverter = CustomAuthenticationConverter()
               }
           }
        }
    }
}
```

12.3.10. Configuring Validation

Using minimal Spring Boot configuration, indicating the authorization server's issuer uri, Resource Server will default to verifying the `iss` claim as well as the `exp` and `nbf` timestamp claims.

In circumstances where validation needs to be customized, Resource Server ships with two standard validators and also accepts custom `OAuth2TokenValidator` instances.

Customizing Timestamp Validation

JWT's typically have a window of validity, with the start of the window indicated in the `nbf` claim and the end indicated in the `exp` claim.

However, every server can experience clock drift, which can cause tokens to appear expired to one server, but not to another. This can cause some implementation heartburn as the number of collaborating servers increases in a distributed system.

Resource Server uses `JwtTimestampValidator` to verify a token's validity window, and it can be configured with a `clockSkew` to alleviate the above problem:

Java

```
@Bean
JwtDecoder jwtDecoder() {
     NimbusJwtDecoder jwtDecoder = (NimbusJwtDecoder)
              JwtDecoders.fromIssuerLocation(issuerUri);

     OAuth2TokenValidator<Jwt> withClockSkew = new
DelegatingOAuth2TokenValidator<>(
             new JwtTimestampValidator(Duration.ofSeconds(60)),
             new JwtIssuerValidator(issuerUri));

     jwtDecoder.setJwtValidator(withClockSkew);

     return jwtDecoder;
}
```

Kotlin

```
@Bean
fun jwtDecoder(): JwtDecoder {
    val jwtDecoder: NimbusJwtDecoder = JwtDecoders.fromIssuerLocation(issuerUri)
as NimbusJwtDecoder

    val withClockSkew: OAuth2TokenValidator<Jwt> = DelegatingOAuth2TokenValidator(
            JwtTimestampValidator(Duration.ofSeconds(60)),
            JwtIssuerValidator(issuerUri))

    jwtDecoder.setJwtValidator(withClockSkew)

    return jwtDecoder
}
```

By default, Resource Server configures a clock skew of 60 seconds.

Configuring a Custom Validator

Adding a check for the `aud` claim is simple with the `OAuth2TokenValidator` API:

Java

```
OAuth2TokenValidator<Jwt> audienceValidator() {
    return new JwtClaimValidator<List<String>>(AUD, aud ->
aud.contains("messaging"));
}
```

Kotlin

```
fun audienceValidator(): OAuth2TokenValidator<Jwt?> {
    return JwtClaimValidator<List<String>>(AUD) { aud -> aud.contains("messaging")
}
}
```

Or, for more control you can implement your own `OAuth2TokenValidator`:

Java

```
static class AudienceValidator implements OAuth2TokenValidator<Jwt> {
    OAuth2Error error = new OAuth2Error("custom_code", "Custom error message",
null);

    @Override
    public OAuth2TokenValidatorResult validate(Jwt jwt) {
        if (jwt.getAudience().contains("messaging")) {
            return OAuth2TokenValidatorResult.success();
        } else {
            return OAuth2TokenValidatorResult.failure(error);
        }
    }
}

// ...

OAuth2TokenValidator<Jwt> audienceValidator() {
    return new AudienceValidator();
}
```

Kotlin

```
internal class AudienceValidator : OAuth2TokenValidator<Jwt> {
    var error: OAuth2Error = OAuth2Error("custom_code", "Custom error message",
null)

    override fun validate(jwt: Jwt): OAuth2TokenValidatorResult {
        return if (jwt.audience.contains("messaging")) {
            OAuth2TokenValidatorResult.success()
        } else {
            OAuth2TokenValidatorResult.failure(error)
        }
    }
}

// ...

fun audienceValidator(): OAuth2TokenValidator<Jwt> {
    return AudienceValidator()
}
```

Then, to add into a resource server, it's a matter of specifying the `JwtDecoder` instance:

Java

```
@Bean
JwtDecoder jwtDecoder() {
    NimbusJwtDecoder jwtDecoder = (NimbusJwtDecoder)
        JwtDecoders.fromIssuerLocation(issuerUri);

    OAuth2TokenValidator<Jwt> audienceValidator = audienceValidator();
    OAuth2TokenValidator<Jwt> withIssuer =
JwtValidators.createDefaultWithIssuer(issuerUri);
    OAuth2TokenValidator<Jwt> withAudience = new
DelegatingOAuth2TokenValidator<>(withIssuer, audienceValidator);

    jwtDecoder.setJwtValidator(withAudience);

    return jwtDecoder;
}
```

Kotlin

```
@Bean
fun jwtDecoder(): JwtDecoder {
    val jwtDecoder: NimbusJwtDecoder = JwtDecoders.fromIssuerLocation(issuerUri)
as NimbusJwtDecoder

    val audienceValidator = audienceValidator()
    val withIssuer: OAuth2TokenValidator<Jwt> =
JwtValidators.createDefaultWithIssuer(issuerUri)
    val withAudience: OAuth2TokenValidator<Jwt> =
DelegatingOAuth2TokenValidator(withIssuer, audienceValidator)

    jwtDecoder.setJwtValidator(withAudience)

    return jwtDecoder
}
```

12.3.11. Configuring Claim Set Mapping

Spring Security uses the Nimbus library for parsing JWTs and validating their signatures. Consequently, Spring Security is subject to Nimbus's interpretation of each field value and how to coerce each into a Java type.

For example, because Nimbus remains Java 7 compatible, it doesn't use `Instant` to represent timestamp fields.

And it's entirely possible to use a different library or for JWT processing, which may make its own coercion decisions that need adjustment.

Or, quite simply, a resource server may want to add or remove claims from a JWT for domain-specific reasons.

For these purposes, Resource Server supports mapping the JWT claim set with `MappedJwtClaimSetConverter`.

Customizing the Conversion of a Single Claim

By default, `MappedJwtClaimSetConverter` will attempt to coerce claims into the following types:

Claim	Java Type
`aud`	`Collection<String>`
`exp`	`Instant`
`iat`	`Instant`
`iss`	`String`
`jti`	`String`
`nbf`	`Instant`
`sub`	`String`

An individual claim's conversion strategy can be configured using `MappedJwtClaimSetConverter.withDefaults`:

Java

```
@Bean
JwtDecoder jwtDecoder() {
    NimbusJwtDecoder jwtDecoder =
NimbusJwtDecoder.withJwkSetUri(jwkSetUri).build();

    MappedJwtClaimSetConverter converter = MappedJwtClaimSetConverter
            .withDefaults(Collections.singletonMap("sub",
this::lookupUserIdBySub));
    jwtDecoder.setClaimSetConverter(converter);

    return jwtDecoder;
}
```

Kotlin

```
@Bean
fun jwtDecoder(): JwtDecoder {
    val jwtDecoder = NimbusJwtDecoder.withJwkSetUri(jwkSetUri).build()

    val converter = MappedJwtClaimSetConverter
            .withDefaults(mapOf("sub" to this::lookupUserIdBySub))
    jwtDecoder.setClaimSetConverter(converter)

    return jwtDecoder
}
```

This will keep all the defaults, except it will override the default claim converter for `sub`.

Adding a Claim

`MappedJwtClaimSetConverter` can also be used to add a custom claim, for example, to adapt to an existing system:

Java

```
MappedJwtClaimSetConverter.withDefaults(Collections.singletonMap("custom", custom
-> "value"));
```

Kotlin

```
MappedJwtClaimSetConverter.withDefaults(mapOf("custom" to Converter<Any, String> {
"value" }))
```

Removing a Claim

And removing a claim is also simple, using the same API:

Java

```
MappedJwtClaimSetConverter.withDefaults(Collections.singletonMap("legacyclaim",
legacy -> null));
```

Kotlin

```
MappedJwtClaimSetConverter.withDefaults(mapOf("legacyclaim" to Converter<Any, Any>
{ null }))
```

Renaming a Claim

In more sophisticated scenarios, like consulting multiple claims at once or renaming a claim, Resource Server accepts any class that implements `Converter<Map<String, Object>, Map<String,Object>>`:

Java

```
public class UsernameSubClaimAdapter implements Converter<Map<String, Object>,
Map<String, Object>> {
    private final MappedJwtClaimSetConverter delegate =
            MappedJwtClaimSetConverter.withDefaults(Collections.emptyMap());

    public Map<String, Object> convert(Map<String, Object> claims) {
        Map<String, Object> convertedClaims = this.delegate.convert(claims);

        String username = (String) convertedClaims.get("user_name");
        convertedClaims.put("sub", username);

        return convertedClaims;
    }
}
```

Kotlin

```
class UsernameSubClaimAdapter : Converter<Map<String, Any?>, Map<String, Any?>> {
    private val delegate =
MappedJwtClaimSetConverter.withDefaults(Collections.emptyMap())
    override fun convert(claims: Map<String, Any?>): Map<String, Any?> {
        val convertedClaims = delegate.convert(claims)
        val username = convertedClaims["user_name"] as String
        convertedClaims["sub"] = username
        return convertedClaims
    }
}
```

And then, the instance can be supplied like normal:

Java

```
@Bean
JwtDecoder jwtDecoder() {
    NimbusJwtDecoder jwtDecoder =
NimbusJwtDecoder.withJwkSetUri(jwkSetUri).build();
    jwtDecoder.setClaimSetConverter(new UsernameSubClaimAdapter());
    return jwtDecoder;
}
```

Kotlin

```
@Bean
fun jwtDecoder(): JwtDecoder {
    val jwtDecoder: NimbusJwtDecoder =
NimbusJwtDecoder.withJwkSetUri(jwkSetUri).build()
    jwtDecoder.setClaimSetConverter(UsernameSubClaimAdapter())
    return jwtDecoder
}
```

12.3.12. Configuring Timeouts

By default, Resource Server uses connection and socket timeouts of 30 seconds each for coordinating with the authorization server.

This may be too short in some scenarios. Further, it doesn't take into account more sophisticated patterns like back-off and discovery.

To adjust the way in which Resource Server connects to the authorization server, `NimbusJwtDecoder` accepts an instance of `RestOperations`:

Java

```
@Bean
public JwtDecoder jwtDecoder(RestTemplateBuilder builder) {
    RestOperations rest = builder
            .setConnectTimeout(Duration.ofSeconds(60))
            .setReadTimeout(Duration.ofSeconds(60))
            .build();

    NimbusJwtDecoder jwtDecoder =
NimbusJwtDecoder.withJwkSetUri(jwkSetUri).restOperations(rest).build();
    return jwtDecoder;
}
```

Kotlin

```
@Bean
fun jwtDecoder(builder: RestTemplateBuilder): JwtDecoder {
    val rest: RestOperations = builder
            .setConnectTimeout(Duration.ofSeconds(60))
            .setReadTimeout(Duration.ofSeconds(60))
            .build()
    return NimbusJwtDecoder.withJwkSetUri(jwkSetUri).restOperations(rest).build()
}
```

Also by default, Resource Server caches in-memory the authorization server's JWK set for 5 minutes, which you may want to adjust. Further, it doesn't take into account more sophisticated caching patterns like eviction or using a shared cache.

To adjust the way in which Resource Server caches the JWK set, `NimbusJwtDecoder` accepts an instance of `Cache`:

Java

```
@Bean
public JwtDecoder jwtDecoder(CacheManager cacheManager) {
    return NimbusJwtDecoder.withJwkSetUri(jwkSetUri)
            .cache(cacheManager.getCache("jwks"))
            .build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(cacheManager: CacheManager): JwtDecoder {
    return NimbusJwtDecoder.withJwkSetUri(jwkSetUri)
            .cache(cacheManager.getCache("jwks"))
            .build()
}
```

When given a `Cache`, Resource Server will use the JWK Set Uri as the key and the JWK Set JSON as the value.

Spring isn't a cache provider, so you'll need to make sure to include the appropriate dependencies, like `spring-boot-starter-cache` and your favorite caching provider.

Whether it's socket or cache timeouts, you may instead want to work with Nimbus directly. To do so, remember that `NimbusJwtDecoder` ships with a constructor that takes Nimbus's `JWTProcessor`.

12.3.13. Minimal Dependencies for Introspection

As described in Minimal Dependencies for JWT most of Resource Server support is collected in `spring-security-oauth2-resource-server`. However unless a custom `OpaqueTokenIntrospector` is provided, the Resource Server will fallback to NimbusOpaqueTokenIntrospector. Meaning that both `spring-security-oauth2-resource-server` and `oauth2-oidc-sdk` are necessary in order to have a working minimal Resource Server that supports opaque Bearer Tokens. Please refer to `spring-security-oauth2-resource-server` in order to determin the correct version for `oauth2-oidc-sdk`.

12.3.14. Minimal Configuration for Introspection

Typically, an opaque token can be verified via an OAuth 2.0 Introspection Endpoint, hosted by the authorization server. This can be handy when revocation is a requirement.

When using Spring Boot, configuring an application as a resource server that uses introspection consists of two basic steps. First, include the needed dependencies and second, indicate the introspection endpoint details.

Specifying the Authorization Server

To specify where the introspection endpoint is, simply do:

```
security:
  oauth2:
    resourceserver:
      opaque-token:
        introspection-uri: https://idp.example.com/introspect
        client-id: client
        client-secret: secret
```

Where `https://idp.example.com/introspect` is the introspection endpoint hosted by your authorization server and `client-id` and `client-secret` are the credentials needed to hit that endpoint.

Resource Server will use these properties to further self-configure and subsequently validate incoming JWTs.

When using introspection, the authorization server's word is the law. If the authorization server responses that the token is valid, then it is.

And that's it!

Startup Expectations

When this property and these dependencies are used, Resource Server will automatically configure itself to validate Opaque Bearer Tokens.

This startup process is quite a bit simpler than for JWTs since no endpoints need to be discovered and no additional validation rules get added.

Runtime Expectations

Once the application is started up, Resource Server will attempt to process any request containing an `Authorization: Bearer` header:

```
GET / HTTP/1.1
Authorization: Bearer some-token-value # Resource Server will process this
```

So long as this scheme is indicated, Resource Server will attempt to process the request according to the Bearer Token specification.

Given an Opaque Token, Resource Server will

1. Query the provided introspection endpoint using the provided credentials and the token
2. Inspect the response for an `{ 'active' : true }` attribute
3. Map each scope to an authority with the prefix `SCOPE_`

The resulting `Authentication#getPrincipal`, by default, is a Spring Security `OAuth2AuthenticatedPrincipal` object, and `Authentication#getName` maps to the token's `sub` property, if one is present.

From here, you may want to jump to:

- How Opaque Token Authentication Works
- Looking Up Attributes Post-Authentication
- Extracting Authorities Manually
- Using Introspection with JWTs

12.3.15. How Opaque Token Authentication Works

Next, let's see the architectural components that Spring Security uses to support opaque token Authentication in servlet-based applications, like the one we just saw.

`OpaqueTokenAuthenticationProvider` is an `AuthenticationProvider` implementation that leverages a `OpaqueTokenIntrospector` to authenticate an opaque token.

Let's take a look at how `OpaqueTokenAuthenticationProvider` works within Spring Security. The figure explains details of how the `AuthenticationManager` in figures from Reading the Bearer Token works.

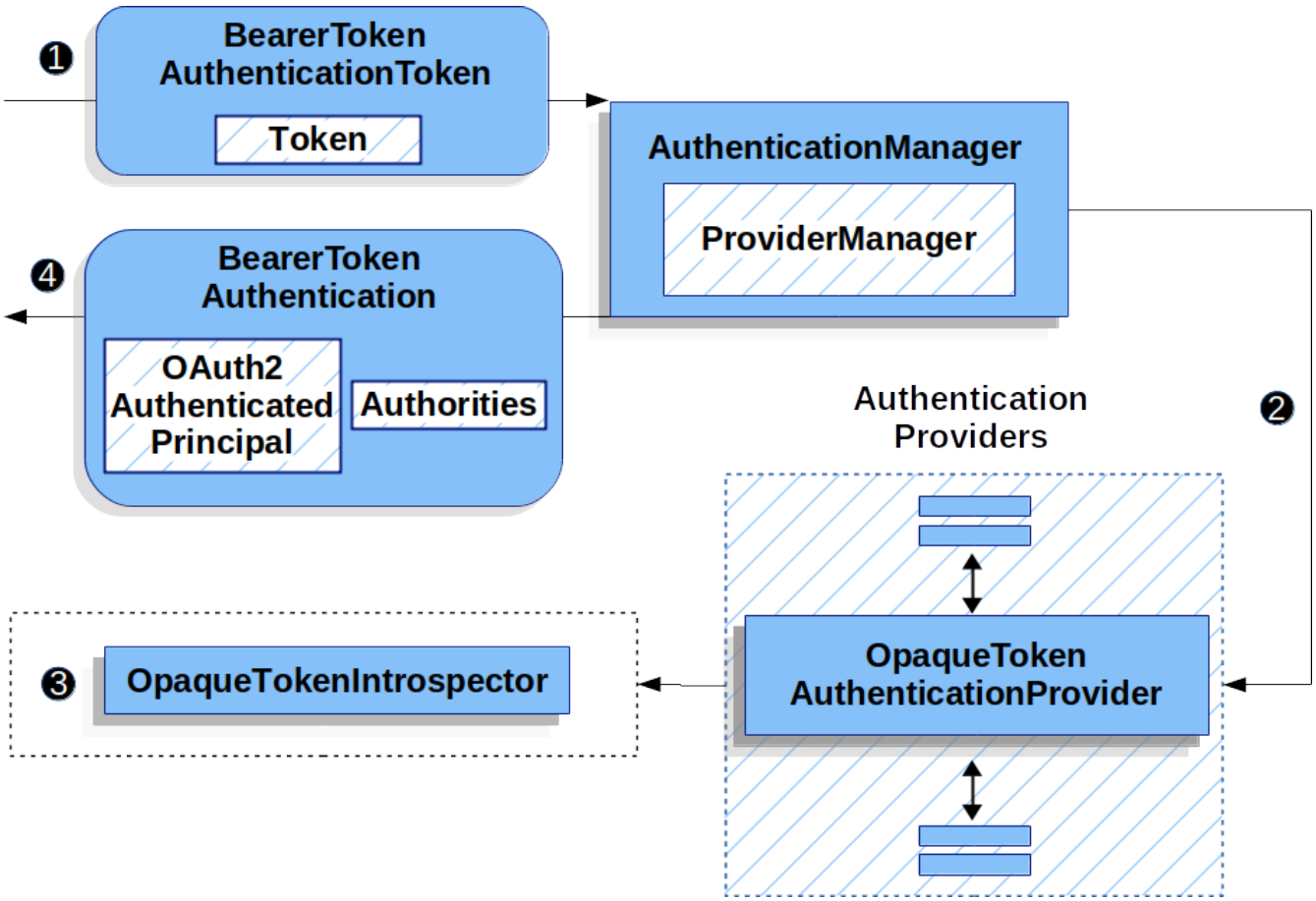


Figure 17. `OpaqueTokenAuthenticationProvider` Usage

1 The authentication `Filter` from Reading the Bearer Token passes a `BearerTokenAuthenticationToken` to the `AuthenticationManager` which is implemented by `ProviderManager`.

(2) The `ProviderManager` is configured to use an AuthenticationProvider of type `OpaqueTokenAuthenticationProvider`.

(3) `OpaqueTokenAuthenticationProvider` introspects the opaque token and adds granted authorities using an `OpaqueTokenIntrospector`. When authentication is successful, the `Authentication` that is returned is of type `BearerTokenAuthentication` and has a principal that is the `OAuth2AuthenticatedPrincipal` returned by the configured `OpaqueTokenIntrospector`. Ultimately, the returned `BearerTokenAuthentication` will be set on the `SecurityContextHolder` by the authentication `Filter`.

12.3.16. Looking Up Attributes Post-Authentication

Once a token is authenticated, an instance of `BearerTokenAuthentication` is set in the `SecurityContext`.

This means that it's available in `@Controller` methods when using `@EnableWebMvc` in your configuration:

Java

```
@GetMapping("/foo")
public String foo(BearerTokenAuthentication authentication) {
    return authentication.getTokenAttributes().get("sub") + " is the subject";
}
```

Kotlin

```
@GetMapping("/foo")
fun foo(authentication: BearerTokenAuthentication): String {
    return authentication.tokenAttributes["sub"].toString() + " is the subject"
}
```

Since `BearerTokenAuthentication` holds an `OAuth2AuthenticatedPrincipal`, that also means that it's available to controller methods, too:

Java

```
@GetMapping("/foo")
public String foo(@AuthenticationPrincipal OAuth2AuthenticatedPrincipal principal)
{
    return principal.getAttribute("sub") + " is the subject";
}
```

Kotlin

```
@GetMapping("/foo")
fun foo(@AuthenticationPrincipal principal: OAuth2AuthenticatedPrincipal): String
{
    return principal.getAttribute<Any>("sub").toString() + " is the subject"
}
```

Looking Up Attributes Via SpEL

Of course, this also means that attributes can be accessed via SpEL.

For example, if using `@EnableGlobalMethodSecurity` so that you can use `@PreAuthorize` annotations, you can do:

Java

```
@PreAuthorize("principal?.attributes['sub'] == 'foo'")
public String forFoosEyesOnly() {
    return "foo";
}
```

Kotlin

```
@PreAuthorize("principal?.attributes['sub'] == 'foo'")
fun forFoosEyesOnly(): String {
    return "foo"
}
```

12.3.17. Overriding or Replacing Boot Auto Configuration

There are two `@Bean` s that Spring Boot generates on Resource Server's behalf.

The first is a `WebSecurityConfigurerAdapter` that configures the app as a resource server. When use Opaque Token, this `WebSecurityConfigurerAdapter` looks like:

Example 130. Default Opaque Token Configuration

Java

```
protected void configure(HttpSecurity http) {
    http
        .authorizeRequests(authorize -> authorize
            .anyRequest().authenticated()
        )
        .oauth2ResourceServer(OAuth2ResourceServerConfigurer::opaqueToken);
}
```

Kotlin

```
override fun configure(http: HttpSecurity) {
    http {
        authorizeRequests {
            authorize(anyRequest, authenticated)
        }
        oauth2ResourceServer {
            opaqueToken { }
        }
    }
}
```

If the application doesn't expose a `WebSecurityConfigurerAdapter` bean, then Spring Boot will expose the above default one.

Replacing this is as simple as exposing the bean within the application:

Example 131. Custom Opaque Token Configuration

Java

```
@EnableWebSecurity
public class MyCustomSecurityConfiguration extends WebSecurityConfigurerAdapter {
    protected void configure(HttpSecurity http) {
        http
            .authorizeRequests(authorize -> authorize
                .mvcMatchers("/messages/**").hasAuthority("SCOPE_message:read")
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .opaqueToken(opaqueToken -> opaqueToken
                    .introspector(myIntrospector())
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class MyCustomSecurityConfiguration : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            authorizeRequests {
                authorize("/messages/**", hasAuthority("SCOPE_message:read"))
                authorize(anyRequest, authenticated)
            }
            oauth2ResourceServer {
                opaqueToken {
                    introspector = myIntrospector()
                }
            }
        }
    }
}
```

The above requires the scope of `message:read` for any URL that starts with `/messages/`.

Methods on the `oauth2ResourceServer` DSL will also override or replace auto configuration.

For example, the second `@Bean` Spring Boot creates is an `OpaqueTokenIntrospector`, which decodes `String` tokens into validated instances of `OAuth2AuthenticatedPrincipal`:

Java

```
@Bean
public OpaqueTokenIntrospector introspector() {
    return new NimbusOpaqueTokenIntrospector(introspectionUri, clientId,
clientSecret);
}
```

Kotlin

```
@Bean
fun introspector(): OpaqueTokenIntrospector {
    return NimbusOpaqueTokenIntrospector(introspectionUri, clientId, clientSecret)
}
```

If the application doesn't expose a `OpaqueTokenIntrospector` bean, then Spring Boot will expose the above default one.

And its configuration can be overridden using `introspectionUri()` and `introspectionClientCredentials()` or replaced using `introspector()`.

Or, if you're not using Spring Boot at all, then both of these components - the filter chain and a `OpaqueTokenIntrospector` can be specified in XML.

The filter chain is specified like so:

Example 132. Default Opaque Token Configuration

Xml

```
<http>
    <intercept-uri pattern="/**" access="authenticated"/>
    <oauth2-resource-server>
        <opaque-token introspector-ref="opaqueTokenIntrospector"/>
    </oauth2-resource-server>
</http>
```

And the `OpaqueTokenIntrospector` like so:

Example 133. Opaque Token Introspector

Xml

```
<bean id="opaqueTokenIntrospector"

class="org.springframework.security.oauth2.server.resource.introspection.NimbusOpa
queTokenIntrospector">
    <constructor-arg
value="${spring.security.oauth2.resourceserver.opaquetoken.introspection_uri}"/>
    <constructor-arg
value="${spring.security.oauth2.resourceserver.opaquetoken.client_id}"/>
    <constructor-arg
value="${spring.security.oauth2.resourceserver.opaquetoken.client_secret}"/>
</bean>
```

Using `introspectionUri()`

An authorization server's Introspection Uri can be configured as a configuration property or it can be supplied in the DSL:

Example 134. Introspection URI Configuration

Java

```
@EnableWebSecurity
public class DirectlyConfiguredIntrospectionUri extends
WebSecurityConfigurerAdapter {
    protected void configure(HttpSecurity http) {
        http
            .authorizeRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .opaqueToken(opaqueToken -> opaqueToken
                    .introspectionUri("https://idp.example.com/introspect")
                    .introspectionClientCredentials("client", "secret")
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class DirectlyConfiguredIntrospectionUri : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            authorizeRequests {
                authorize(anyRequest, authenticated)
            }
            oauth2ResourceServer {
                opaqueToken {
                    introspectionUri = "https://idp.example.com/introspect"
                    introspectionClientCredentials("client", "secret")
                }
            }
        }
    }
}
```

Xml

```
<bean id="opaqueTokenIntrospector"

class="org.springframework.security.oauth2.server.resource.introspection.NimbusOpa
queTokenIntrospector">
    <constructor-arg value="https://idp.example.com/introspect"/>
    <constructor-arg value="client"/>
    <constructor-arg value="secret"/>
</bean>
```

Using `introspectionUri()` takes precedence over any configuration property.

Using `introspector()`

More powerful than `introspectionUri()` is `introspector()`, which will completely replace any Boot auto configuration of `OpaqueTokenIntrospector`:

Example 135. Introspector Configuration

Java

```
@EnableWebSecurity
public class DirectlyConfiguredIntrospector extends WebSecurityConfigurerAdapter {
    protected void configure(HttpSecurity http) {
        http
            .authorizeRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .opaqueToken(opaqueToken -> opaqueToken
                    .introspector(myCustomIntrospector())
                )
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class DirectlyConfiguredIntrospector : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            authorizeRequests {
                authorize(anyRequest, authenticated)
            }
            oauth2ResourceServer {
                opaqueToken {
                    introspector = myCustomIntrospector()
                }
            }
        }
    }
}
```

Xml

```
<http>
    <intercept-uri pattern="/**" access="authenticated"/>
    <oauth2-resource-server>
        <opaque-token introspector-ref="myCustomIntrospector"/>
    </oauth2-resource-server>
</http>
```

This is handy when deeper configuration, like authority mapping, JWT revocation, or request timeouts, is necessary.

Exposing a `OpaqueTokenIntrospector` `@Bean`

Or, exposing a `OpaqueTokenIntrospector` `@Bean` has the same effect as `introspector()`:

```
@Bean
public OpaqueTokenIntrospector introspector() {
    return new NimbusOpaqueTokenIntrospector(introspectionUri, clientId,
clientSecret);
}
```

12.3.18. Configuring Authorization

An OAuth 2.0 Introspection endpoint will typically return a `scope` attribute, indicating the scopes (or authorities) it's been granted, for example:

```
{ …, "scope" : "messages contacts"}
```

When this is the case, Resource Server will attempt to coerce these scopes into a list of granted authorities, prefixing each scope with the string "SCOPE_".

This means that to protect an endpoint or method with a scope derived from an Opaque Token, the corresponding expressions should include this prefix:

Example 136. Authorization Opaque Token Configuration

Java

```
@EnableWebSecurity
public class MappedAuthorities extends WebSecurityConfigurerAdapter {
    protected void configure(HttpSecurity http) {
        http
            .authorizeRequests(authorizeRequests -> authorizeRequests
                .mvcMatchers("/contacts/**").hasAuthority("SCOPE_contacts")
                .mvcMatchers("/messages/**").hasAuthority("SCOPE_messages")
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(OAuth2ResourceServerConfigurer::opaqueToken);
    }
}
```

Kotlin

```
@EnableWebSecurity
class MappedAuthorities : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
       http {
            authorizeRequests {
                authorize("/contacts/**", hasAuthority("SCOPE_contacts"))
                authorize("/messages/**", hasAuthority("SCOPE_messages"))
                authorize(anyRequest, authenticated)
            }
           oauth2ResourceServer {
               opaqueToken { }
           }
        }
    }
}
```

Xml

```
<http>
    <intercept-uri pattern="/contacts/**"
access="hasAuthority('SCOPE_contacts')"/>
    <intercept-uri pattern="/messages/**"
access="hasAuthority('SCOPE_messages')"/>
    <oauth2-resource-server>
        <opaque-token introspector-ref="opaqueTokenIntrospector"/>
    </oauth2-resource-server>
</http>
```

Or similarly with method security:

Java

```
@PreAuthorize("hasAuthority('SCOPE_messages')")
public List<Message> getMessages(...) {}
```

Kotlin

```
@PreAuthorize("hasAuthority('SCOPE_messages')")
fun getMessages(): List<Message?> {}
```

Extracting Authorities Manually

By default, Opaque Token support will extract the scope claim from an introspection response and parse it into individual `GrantedAuthority` instances.

For example, if the introspection response were:

```
{
    "active" : true,
    "scope" : "message:read message:write"
}
```

Then Resource Server would generate an `Authentication` with two authorities, one for `message:read` and the other for `message:write`.

This can, of course, be customized using a custom `OpaqueTokenIntrospector` that takes a look at the attribute set and converts in its own way:

Java

```
public class CustomAuthoritiesOpaqueTokenIntrospector implements
OpaqueTokenIntrospector {
    private OpaqueTokenIntrospector delegate =
            new
NimbusOpaqueTokenIntrospector("https://idp.example.org/introspect", "client",
"secret");

    public OAuth2AuthenticatedPrincipal introspect(String token) {
        OAuth2AuthenticatedPrincipal principal = this.delegate.introspect(token);
        return new DefaultOAuth2AuthenticatedPrincipal(
                principal.getName(), principal.getAttributes(),
extractAuthorities(principal));
    }

    private Collection<GrantedAuthority>
extractAuthorities(OAuth2AuthenticatedPrincipal principal) {
        List<String> scopes =
principal.getAttribute(OAuth2IntrospectionClaimNames.SCOPE);
        return scopes.stream()
                .map(SimpleGrantedAuthority::new)
                .collect(Collectors.toList());
    }
}
```

Kotlin

```
class CustomAuthoritiesOpaqueTokenIntrospector : OpaqueTokenIntrospector {
    private val delegate: OpaqueTokenIntrospector =
NimbusOpaqueTokenIntrospector("https://idp.example.org/introspect", "client",
"secret")
    override fun introspect(token: String): OAuth2AuthenticatedPrincipal {
        val principal: OAuth2AuthenticatedPrincipal = delegate.introspect(token)
        return DefaultOAuth2AuthenticatedPrincipal(
                principal.name, principal.attributes,
extractAuthorities(principal))
    }

    private fun extractAuthorities(principal: OAuth2AuthenticatedPrincipal):
Collection<GrantedAuthority> {
        val scopes: List<String> =
principal.getAttribute(OAuth2IntrospectionClaimNames.SCOPE)
        return scopes
                .map { SimpleGrantedAuthority(it) }
    }
}
```

Thereafter, this custom introspector can be configured simply by exposing it as a `@Bean`:

Java

```
@Bean
public OpaqueTokenIntrospector introspector() {
    return new CustomAuthoritiesOpaqueTokenIntrospector();
}
```

Kotlin

```
@Bean
fun introspector(): OpaqueTokenIntrospector {
    return CustomAuthoritiesOpaqueTokenIntrospector()
}
```

12.3.19. Configuring Timeouts

By default, Resource Server uses connection and socket timeouts of 30 seconds each for coordinating with the authorization server.

This may be too short in some scenarios. Further, it doesn't take into account more sophisticated patterns like back-off and discovery.

To adjust the way in which Resource Server connects to the authorization server, `NimbusOpaqueTokenIntrospector` accepts an instance of `RestOperations`:

Java

```
@Bean
public OpaqueTokenIntrospector introspector(RestTemplateBuilder builder,
OAuth2ResourceServerProperties properties) {
    RestOperations rest = builder
            .basicAuthentication(properties.getOpaquetoken().getClientId(),
properties.getOpaquetoken().getClientSecret())
            .setConnectTimeout(Duration.ofSeconds(60))
            .setReadTimeout(Duration.ofSeconds(60))
            .build();

    return new NimbusOpaqueTokenIntrospector(introspectionUri, rest);
}
```

Kotlin

```
@Bean
fun introspector(builder: RestTemplateBuilder, properties:
OAuth2ResourceServerProperties): OpaqueTokenIntrospector? {
    val rest: RestOperations = builder
            .basicAuthentication(properties.opaquetoken.clientId,
properties.opaquetoken.clientSecret)
            .setConnectTimeout(Duration.ofSeconds(60))
            .setReadTimeout(Duration.ofSeconds(60))
            .build()
    return NimbusOpaqueTokenIntrospector(introspectionUri, rest)
}
```

12.3.20. Using Introspection with JWTs

A common question is whether or not introspection is compatible with JWTs. Spring Security's Opaque Token support has been designed to not care about the format of the token — it will gladly pass any token to the introspection endpoint provided.

So, let's say that you've got a requirement that requires you to check with the authorization server on each request, in case the JWT has been revoked.

Even though you are using the JWT format for the token, your validation method is introspection, meaning you'd want to do:

```
spring:
  security:
    oauth2:
      resourceserver:
        opaque-token:
          introspection-uri: https://idp.example.org/introspection
          client-id: client
          client-secret: secret
```

In this case, the resulting `Authentication` would be `BearerTokenAuthentication`. Any attributes in the corresponding `OAuth2AuthenticatedPrincipal` would be whatever was returned by the introspection endpoint.

But, let's say that, oddly enough, the introspection endpoint only returns whether or not the token is active. Now what?

In this case, you can create a custom `OpaqueTokenIntrospector` that still hits the endpoint, but then updates the returned principal to have the JWTs claims as the attributes:

Java

```
public class JwtOpaqueTokenIntrospector implements OpaqueTokenIntrospector {
    private OpaqueTokenIntrospector delegate =
            new
NimbusOpaqueTokenIntrospector("https://idp.example.org/introspect", "client",
"secret");
    private JwtDecoder jwtDecoder = new NimbusJwtDecoder(new
ParseOnlyJWTProcessor());

    public OAuth2AuthenticatedPrincipal introspect(String token) {
        OAuth2AuthenticatedPrincipal principal = this.delegate.introspect(token);
        try {
            Jwt jwt = this.jwtDecoder.decode(token);
            return new DefaultOAuth2AuthenticatedPrincipal(jwt.getClaims(),
NO_AUTHORITIES);
        } catch (JwtException ex) {
            throw new OAuth2IntrospectionException(ex);
        }
    }

    private static class ParseOnlyJWTProcessor extends
DefaultJWTProcessor<SecurityContext> {
        JWTClaimsSet process(SignedJWT jwt, SecurityContext context)
                throws JOSEException {
            return jwt.getJWTClaimsSet();
        }
    }
}
```

Kotlin

```
class JwtOpaqueTokenIntrospector : OpaqueTokenIntrospector {
    private val delegate: OpaqueTokenIntrospector =
NimbusOpaqueTokenIntrospector("https://idp.example.org/introspect", "client",
"secret")
    private val jwtDecoder: JwtDecoder = NimbusJwtDecoder(ParseOnlyJWTProcessor())
    override fun introspect(token: String): OAuth2AuthenticatedPrincipal {
        val principal = delegate.introspect(token)
        return try {
            val jwt: Jwt = jwtDecoder.decode(token)
            DefaultOAuth2AuthenticatedPrincipal(jwt.claims, NO_AUTHORITIES)
        } catch (ex: JwtException) {
            throw OAuth2IntrospectionException(ex.message)
        }
    }

    private class ParseOnlyJWTProcessor : DefaultJWTProcessor<SecurityContext>() {
        override fun process(jwt: SignedJWT, context: SecurityContext):
JWTClaimsSet {
            return jwt.jwtClaimsSet
        }
    }
}
```

Thereafter, this custom introspector can be configured simply by exposing it as a `@Bean`:

Java

```
@Bean
public OpaqueTokenIntrospector introspector() {
    return new JwtOpaqueTokenIntrospector();
}
```

Kotlin

```
@Bean
fun introspector(): OpaqueTokenIntrospector {
    return JwtOpaqueTokenIntrospector()
}
```

12.3.21. Calling a `/userinfo` Endpoint

Generally speaking, a Resource Server doesn't care about the underlying user, but instead about the authorities that have been granted.

That said, at times it can be valuable to tie the authorization statement back to a user.

If an application is also using `spring-security-oauth2-client`, having set up the appropriate `ClientRegistrationRepository`, then this is quite simple with a custom `OpaqueTokenIntrospector`. This implementation below does three things:

- Delegates to the introspection endpoint, to affirm the token's validity
- Looks up the appropriate client registration associated with the `/userinfo` endpoint
- Invokes and returns the response from the `/userinfo` endpoint

Java

```
public class UserInfoOpaqueTokenIntrospector implements OpaqueTokenIntrospector {
    private final OpaqueTokenIntrospector delegate =
            new
NimbusOpaqueTokenIntrospector("https://idp.example.org/introspect", "client",
"secret");
    private final OAuth2UserService oauth2UserService = new
DefaultOAuth2UserService();

    private final ClientRegistrationRepository repository;

    // ... constructor

    @Override
    public OAuth2AuthenticatedPrincipal introspect(String token) {
        OAuth2AuthenticatedPrincipal authorized = this.delegate.introspect(token);
        Instant issuedAt = authorized.getAttribute(ISSUED_AT);
        Instant expiresAt = authorized.getAttribute(EXPIRES_AT);
        ClientRegistration clientRegistration =
this.repository.findByRegistrationId("registration-id");
        OAuth2AccessToken token = new OAuth2AccessToken(BEARER, token, issuedAt,
expiresAt);
        OAuth2UserRequest oauth2UserRequest = new
OAuth2UserRequest(clientRegistration, token);
        return this.oauth2UserService.loadUser(oauth2UserRequest);
    }
}
```

Kotlin

```
class UserInfoOpaqueTokenIntrospector : OpaqueTokenIntrospector {
    private val delegate: OpaqueTokenIntrospector =
NimbusOpaqueTokenIntrospector("https://idp.example.org/introspect", "client",
"secret")
    private val oauth2UserService = DefaultOAuth2UserService()
    private val repository: ClientRegistrationRepository? = null

    // ... constructor

    override fun introspect(token: String): OAuth2AuthenticatedPrincipal {
        val authorized = delegate.introspect(token)
        val issuedAt: Instant? = authorized.getAttribute(ISSUED_AT)
        val expiresAt: Instant? = authorized.getAttribute(EXPIRES_AT)
        val clientRegistration: ClientRegistration =
repository!!.findByRegistrationId("registration-id")
        val accessToken = OAuth2AccessToken(BEARER, token, issuedAt, expiresAt)
        val oauth2UserRequest = OAuth2UserRequest(clientRegistration, accessToken)
        return oauth2UserService.loadUser(oauth2UserRequest)
    }
}
```

If you aren't using `spring-security-oauth2-client`, it's still quite simple. You will simply need to invoke the `/userinfo` with your own instance of `WebClient`:

Java

```
public class UserInfoOpaqueTokenIntrospector implements OpaqueTokenIntrospector {
    private final OpaqueTokenIntrospector delegate =
            new
NimbusOpaqueTokenIntrospector("https://idp.example.org/introspect", "client",
"secret");
    private final WebClient rest = WebClient.create();

    @Override
    public OAuth2AuthenticatedPrincipal introspect(String token) {
        OAuth2AuthenticatedPrincipal authorized = this.delegate.introspect(token);
        return makeUserInfoRequest(authorized);
    }
}
```

Kotlin

```
class UserInfoOpaqueTokenIntrospector : OpaqueTokenIntrospector {
    private val delegate: OpaqueTokenIntrospector =
NimbusOpaqueTokenIntrospector("https://idp.example.org/introspect", "client",
"secret")
    private val rest: WebClient = WebClient.create()

    override fun introspect(token: String): OAuth2AuthenticatedPrincipal {
        val authorized = delegate.introspect(token)
        return makeUserInfoRequest(authorized)
    }
}
```

Either way, having created your `OpaqueTokenIntrospector`, you should publish it as a `@Bean` to override the defaults:

Java

```
@Bean
OpaqueTokenIntrospector introspector() {
    return new UserInfoOpaqueTokenIntrospector(...);
}
```

Kotlin

```
@Bean
fun introspector(): OpaqueTokenIntrospector {
    return UserInfoOpaqueTokenIntrospector(...)
}
```

12.3.22. Supporting both JWT and Opaque Token

In some cases, you may have a need to access both kinds of tokens. For example, you may support more than one tenant where one tenant issues JWTs and the other issues opaque tokens.

If this decision must be made at request-time, then you can use an `AuthenticationManagerResolver` to achieve it, like so:

Java

```
@Bean
AuthenticationManagerResolver<HttpServletRequest>
tokenAuthenticationManagerResolver() {
    BearerTokenResolver bearerToken = new DefaultBearerTokenResolver();
    JwtAuthenticationProvider jwt = jwt();
    OpaqueTokenAuthenticationProvider opaqueToken = opaqueToken();

    return request -> {
        if (useJwt(request)) {
            return jwt::authenticate;
        } else {
            return opaqueToken::authenticate;
        }
    }
}
```

Kotlin

```
@Bean
fun tokenAuthenticationManagerResolver():
AuthenticationManagerResolver<HttpServletRequest> {
    val bearerToken: BearerTokenResolver = DefaultBearerTokenResolver()
    val jwt: JwtAuthenticationProvider = jwt()
    val opaqueToken: OpaqueTokenAuthenticationProvider = opaqueToken()

    return AuthenticationManagerResolver { request ->
        if (useJwt(request)) {
            AuthenticationManager { jwt.authenticate(it) }
        } else {
            AuthenticationManager { opaqueToken.authenticate(it) }
        }
    }
}
```

The implementation of `useJwt(HttpServletRequest)` will likely depend on custom request material like the path.

And then specify this `AuthenticationManagerResolver` in the DSL:

Example 137. Authentication Manager Resolver

Java

```
http
    .authorizeRequests(authorize -> authorize
        .anyRequest().authenticated()
    )
    .oauth2ResourceServer(oauth2 -> oauth2
        .authenticationManagerResolver(this.tokenAuthenticationManagerResolver)
    );
```

Kotlin

```
http {
    authorizeRequests {
        authorize(anyRequest, authenticated)
    }
    oauth2ResourceServer {
        authenticationManagerResolver = tokenAuthenticationManagerResolver()
    }
}
```

Xml

```
<http>
    <oauth2-resource-server authentication-manager-resolver-
ref="tokenAuthenticationManagerResolver"/>
</http>
```

12.3.23. Multi-tenancy

A resource server is considered multi-tenant when there are multiple strategies for verifying a bearer token, keyed by some tenant identifier.

For example, your resource server may accept bearer tokens from two different authorization servers. Or, your authorization server may represent a multiplicity of issuers.

In each case, there are two things that need to be done and trade-offs associated with how you choose to do them:

1. Resolve the tenant
2. Propagate the tenant

Resolving the Tenant By Claim

One way to differentiate tenants is by the issuer claim. Since the issuer claim accompanies signed JWTs, this can be done with the `JwtIssuerAuthenticationManagerResolver`, like so:

Example 138. Multitenancy Tenant by JWT Claim

Java

```
JwtIssuerAuthenticationManagerResolver authenticationManagerResolver = new
JwtIssuerAuthenticationManagerResolver
    ("https://idp.example.org/issuerOne", "https://idp.example.org/issuerTwo");

http
    .authorizeRequests(authorize -> authorize
        .anyRequest().authenticated()
    )
    .oauth2ResourceServer(oauth2 -> oauth2
        .authenticationManagerResolver(authenticationManagerResolver)
    );
```

Kotlin

```
val customAuthenticationManagerResolver = JwtIssuerAuthenticationManagerResolver
    ("https://idp.example.org/issuerOne", "https://idp.example.org/issuerTwo")
http {
    authorizeRequests {
        authorize(anyRequest, authenticated)
    }
    oauth2ResourceServer {
        authenticationManagerResolver = customAuthenticationManagerResolver
    }
}
```

Xml

```
<http>
    <oauth2-resource-server authentication-manager-resolver-
ref="authenticationManagerResolver"/>
</http>

<bean id="authenticationManagerResolver"

class="org.springframework.security.oauth2.server.resource.authentication.JwtIssue
rAuthenticationManagerResolver">
    <constructor-arg>
        <list>
            <value>https://idp.example.org/issuerOne</value>
            <value>https://idp.example.org/issuerTwo</value>
        </list>
    </constructor-arg>
</bean>
```

This is nice because the issuer endpoints are loaded lazily. In fact, the corresponding `JwtAuthenticationProvider` is instantiated only when the first request with the corresponding issuer is sent. This allows for an application startup that is independent from those authorization servers being up and available.

Dynamic Tenants

Of course, you may not want to restart the application each time a new tenant is added. In this case, you can configure the `JwtIssuerAuthenticationManagerResolver` with a repository of `AuthenticationManager` instances, which you can edit at runtime, like so:

Java

```
private void addManager(Map<String, AuthenticationManager> authenticationManagers,
String issuer) {
    JwtAuthenticationProvider authenticationProvider = new
JwtAuthenticationProvider
            (JwtDecoders.fromIssuerLocation(issuer));
    authenticationManagers.put(issuer, authenticationProvider::authenticate);
}

// ...

JwtIssuerAuthenticationManagerResolver authenticationManagerResolver =
        new JwtIssuerAuthenticationManagerResolver(authenticationManagers::get);

http
    .authorizeRequests(authorize -> authorize
        .anyRequest().authenticated()
    )
    .oauth2ResourceServer(oauth2 -> oauth2
        .authenticationManagerResolver(authenticationManagerResolver)
    );
```

Kotlin

```
private fun addManager(authenticationManagers: MutableMap<String,
AuthenticationManager>, issuer: String) {
    val authenticationProvider =
JwtAuthenticationProvider(JwtDecoders.fromIssuerLocation(issuer))
    authenticationManagers[issuer] = AuthenticationManager {
        authentication: Authentication? ->
authenticationProvider.authenticate(authentication)
    }
}

// ...

val customAuthenticationManagerResolver: JwtIssuerAuthenticationManagerResolver =
    JwtIssuerAuthenticationManagerResolver(authenticationManagers::get)
http {
    authorizeRequests {
        authorize(anyRequest, authenticated)
    }
    oauth2ResourceServer {
        authenticationManagerResolver = customAuthenticationManagerResolver
    }
}
```

In this case, you construct `JwtIssuerAuthenticationManagerResolver` with a strategy for obtaining the `AuthenticationManager` given the issuer. This approach allows us to add and remove elements from the repository (shown as a `Map` in the snippet) at runtime.

It would be unsafe to simply take any issuer and construct an `AuthenticationManager` from it. The issuer should be one that the code can verify from a trusted source like a list of allowed issuers.

Parsing the Claim Only Once

You may have observed that this strategy, while simple, comes with the trade-off that the JWT is parsed once by the `AuthenticationManagerResolver` and then again by the `JwtDecoder` later on in the request.

This extra parsing can be alleviated by configuring the `JwtDecoder` directly with a `JWTClaimsSetAwareJWSKeySelector` from Nimbus:

Java

```
@Component
public class TenantJWSKeySelector
    implements JWTClaimsSetAwareJWSKeySelector<SecurityContext> {

    private final TenantRepository tenants; ①
    private final Map<String, JWSKeySelector<SecurityContext>> selectors = new
ConcurrentHashMap<>(); ②

    public TenantJWSKeySelector(TenantRepository tenants) {
        this.tenants = tenants;
    }

    @Override
    public List<? extends Key> selectKeys(JWSHeader jwsHeader, JWTClaimsSet
jwtClaimsSet, SecurityContext securityContext)
            throws KeySourceException {
        return this.selectors.computeIfAbsent(toTenant(jwtClaimsSet),
this::fromTenant)
                .selectJWSKeys(jwsHeader, securityContext);
    }

    private String toTenant(JWTClaimsSet claimSet) {
        return (String) claimSet.getClaim("iss");
    }

    private JWSKeySelector<SecurityContext> fromTenant(String tenant) {
        return Optional.ofNullable(this.tenantRepository.findById(tenant)) ③
                .map(t -> t.getAttrbute("jwks_uri"))
                .map(this::fromUri)
                .orElseThrow(() -> new IllegalArgumentException("unknown
tenant"));
    }

    private JWSKeySelector<SecurityContext> fromUri(String uri) {
        try {
            return JWSAlgorithmFamilyJWSKeySelector.fromJWKSetURL(new URL(uri));
④
        } catch (Exception ex) {
            throw new IllegalArgumentException(ex);
        }
    }
}
```

Kotlin

```
@Component
class TenantJWSKeySelector(tenants: TenantRepository) :
JWTClaimsSetAwareJWSKeySelector<SecurityContext> {
    private val tenants: TenantRepository ①
    private val selectors: MutableMap<String, JWSKeySelector<SecurityContext>> =
ConcurrentHashMap() ②

    init {
        this.tenants = tenants
    }

    fun selectKeys(jwsHeader: JWSHeader?, jwtClaimsSet: JWTClaimsSet,
securityContext: SecurityContext): List<Key?> {
        return selectors.computeIfAbsent(toTenant(jwtClaimsSet)) { tenant: String
-> fromTenant(tenant) }
                .selectJWSKeys(jwsHeader, securityContext)
    }

    private fun toTenant(claimSet: JWTClaimsSet): String {
        return claimSet.getClaim("iss") as String
    }

    private fun fromTenant(tenant: String): JWSKeySelector<SecurityContext> {
        return Optional.ofNullable(this.tenants.findById(tenant)) ③
                .map { t -> t.getAttrbute("jwks_uri") }
                .map { uri: String -> fromUri(uri) }
                .orElseThrow { IllegalArgumentException("unknown tenant") }
    }

    private fun fromUri(uri: String): JWSKeySelector<SecurityContext?> {
        return try {
            JWSAlgorithmFamilyJWSKeySelector.fromJWKSetURL(URL(uri)) ④
        } catch (ex: Exception) {
            throw IllegalArgumentException(ex)
        }
    }
}
```

① A hypothetical source for tenant information

② A cache for `JWKKeySelector`s, keyed by tenant identifier

③ Looking up the tenant is more secure than simply calculating the JWK Set endpoint on the fly - the lookup acts as a list of allowed tenants

④ Create a `JWSKeySelector` via the types of keys that come back from the JWK Set endpoint - the lazy lookup here means that you don't need to configure all tenants at startup

The above key selector is a composition of many key selectors. It chooses which key selector to use

based on the `iss` claim in the JWT.

To use this approach, make sure that the authorization server is configured to include the claim set as part of the token's signature. Without this, you have no guarantee that the issuer hasn't been altered by a bad actor.

Next, we can construct a `JWTProcessor`:

Java

```
@Bean
JWTProcessor jwtProcessor(JWTClaimSetJWSKeySelector keySelector) {
    ConfigurableJWTProcessor<SecurityContext> jwtProcessor =
            new DefaultJWTProcessor();
    jwtProcessor.setJWTClaimSetJWSKeySelector(keySelector);
    return jwtProcessor;
}
```

Kotlin

```
@Bean
fun jwtProcessor(keySelector: JWTClaimsSetAwareJWSKeySelector<SecurityContext>):
JWTProcessor<SecurityContext> {
    val jwtProcessor = DefaultJWTProcessor<SecurityContext>()
    jwtProcessor.jwtClaimsSetAwareJWSKeySelector = keySelector
    return jwtProcessor
}
```

As you are already seeing, the trade-off for moving tenant-awareness down to this level is more configuration. We have just a bit more.

Next, we still want to make sure you are validating the issuer. But, since the issuer may be different per JWT, then you'll need a tenant-aware validator, too:

Java

```
@Component
public class TenantJwtIssuerValidator implements OAuth2TokenValidator<Jwt> {
    private final TenantRepository tenants;
    private final Map<String, JwtIssuerValidator> validators = new
ConcurrentHashMap<>();

    public TenantJwtIssuerValidator(TenantRepository tenants) {
        this.tenants = tenants;
    }

    @Override
    public OAuth2TokenValidatorResult validate(Jwt token) {
        return this.validators.computeIfAbsent(toTenant(token), this::fromTenant)
                .validate(token);
    }

    private String toTenant(Jwt jwt) {
        return jwt.getIssuer();
    }

    private JwtIssuerValidator fromTenant(String tenant) {
        return Optional.ofNullable(this.tenants.findById(tenant))
                .map(t -> t.getAttribute("issuer"))
                .map(JwtIssuerValidator::new)
                .orElseThrow(() -> new IllegalArgumentException("unknown
tenant"));
    }
}
```

Kotlin

```
@Component
class TenantJwtIssuerValidator(tenants: TenantRepository) :
OAuth2TokenValidator<Jwt> {
    private val tenants: TenantRepository
    private val validators: MutableMap<String, JwtIssuerValidator> =
ConcurrentHashMap()
    override fun validate(token: Jwt): OAuth2TokenValidatorResult {
        return validators.computeIfAbsent(toTenant(token)) { tenant: String ->
fromTenant(tenant) }
                .validate(token)
    }

    private fun toTenant(jwt: Jwt): String {
        return jwt.issuer.toString()
    }

    private fun fromTenant(tenant: String): JwtIssuerValidator {
        return Optional.ofNullable(tenants.findById(tenant))
                .map({ t -> t.getAttribute("issuer") })
                .map({ JwtIssuerValidator() })
                .orElseThrow({ IllegalArgumentException("unknown tenant") })
    }

    init {
        this.tenants = tenants
    }
}
```

Now that we have a tenant-aware processor and a tenant-aware validator, we can proceed with creating our `JwtDecoder`:

Java

```
@Bean
JwtDecoder jwtDecoder(JWTProcessor jwtProcessor, OAuth2TokenValidator<Jwt>
jwtValidator) {
    NimbusJwtDecoder decoder = new NimbusJwtDecoder(processor);
    OAuth2TokenValidator<Jwt> validator = new DelegatingOAuth2TokenValidator<>
            (JwtValidators.createDefault(), this.jwtValidator);
    decoder.setJwtValidator(validator);
    return decoder;
}
```

Kotlin

```
@Bean
fun jwtDecoder(jwtProcessor: JWTProcessor<SecurityContext>?, jwtValidator:
OAuth2TokenValidator<Jwt>?): JwtDecoder {
    val decoder = NimbusJwtDecoder(jwtProcessor)
    val validator: OAuth2TokenValidator<Jwt> =
DelegatingOAuth2TokenValidator(JwtValidators.createDefault(), jwtValidator)
    decoder.setJwtValidator(validator)
    return decoder
}
```

We've finished talking about resolving the tenant.

If you've chosen to resolve the tenant by something other than a JWT claim, then you'll need to make sure you address your downstream resource servers in the same way. For example, if you are resolving it by subdomain, you may need to address the downstream resource server using the same subdomain.

However, if you resolve it by a claim in the bearer token, read on to learn about Spring Security's support for bearer token propagation.

12.3.24. Bearer Token Resolution

By default, Resource Server looks for a bearer token in the `Authorization` header. This, however, can be customized in a handful of ways.

Reading the Bearer Token from a Custom Header

For example, you may have a need to read the bearer token from a custom header. To achieve this, you can expose a `DefaultBearerTokenResolver` as a bean, or wire an instance into the DSL, as you can see in the following example:

Example 139. Custom Bearer Token Header

Java

```
@Bean
BearerTokenResolver bearerTokenResolver() {
    DefaultBearerTokenResolver bearerTokenResolver = new
DefaultBearerTokenResolver();
    bearerTokenResolver.setBearerTokenHeaderName(HttpHeaders.PROXY_AUTHORIZATION);
    return bearerTokenResolver;
}
```

Kotlin

```
@Bean
fun bearerTokenResolver(): BearerTokenResolver {
    val bearerTokenResolver = DefaultBearerTokenResolver()
    bearerTokenResolver.setBearerTokenHeaderName(HttpHeaders.PROXY_AUTHORIZATION)
    return bearerTokenResolver
}
```

Xml

```
<http>
    <oauth2-resource-server bearer-token-resolver-ref="bearerTokenResolver"/>
</http>

<bean id="bearerTokenResolver"

class="org.springframework.security.oauth2.server.resource.web.DefaultBearerTokenR
esolver">
    <property name="bearerTokenHeaderName" value="Proxy-Authorization"/>
</bean>
```

Or, in circumstances where a provider is using both a custom header and value, you can use `HeaderBearerTokenResolver` instead.

Reading the Bearer Token from a Form Parameter

Or, you may wish to read the token from a form parameter, which you can do by configuring the `DefaultBearerTokenResolver`, as you can see below:

Example 140. Form Parameter Bearer Token

Java

```
DefaultBearerTokenResolver resolver = new DefaultBearerTokenResolver();
resolver.setAllowFormEncodedBodyParameter(true);
http
    .oauth2ResourceServer(oauth2 -> oauth2
        .bearerTokenResolver(resolver)
    );
```

Kotlin

```
val resolver = DefaultBearerTokenResolver()
resolver.setAllowFormEncodedBodyParameter(true)
http {
    oauth2ResourceServer {
        bearerTokenResolver = resolver
    }
}
```

Xml

```
<http>
    <oauth2-resource-server bearer-token-resolver-ref="bearerTokenResolver"/>
</http>

<bean id="bearerTokenResolver"

class="org.springframework.security.oauth2.server.resource.web.HeaderBearerTokenRe
solver">
    <property name="allowFormEncodedBodyParameter" value="true"/>
</bean>
```

12.3.25. Bearer Token Propagation

Now that you're resource server has validated the token, it might be handy to pass it to downstream services. This is quite simple with `ServletBearerExchangeFilterFunction`, which you can see in the following example:

Java

```
@Bean
public WebClient rest() {
    return WebClient.builder()
            .filter(new ServletBearerExchangeFilterFunction())
            .build();
}
```

Kotlin

```
@Bean
fun rest(): WebClient {
    return WebClient.builder()
            .filter(ServletBearerExchangeFilterFunction())
            .build()
}
```

When the above `WebClient` is used to perform requests, Spring Security will look up the current `Authentication` and extract any `AbstractOAuth2Token` credential. Then, it will propagate that token in the `Authorization` header.

For example:

Java

```
this.rest.get()
        .uri("https://other-service.example.com/endpoint")
        .retrieve()
        .bodyToMono(String.class)
        .block()
```

Kotlin

```
this.rest.get()
        .uri("https://other-service.example.com/endpoint")
        .retrieve()
        .bodyToMono<String>()
        .block()
```

Will invoke the `https://other-service.example.com/endpoint`, adding the bearer token `Authorization` header for you.

In places where you need to override this behavior, it's a simple matter of supplying the header yourself, like so:

Java

```
this.rest.get()
        .uri("https://other-service.example.com/endpoint")
        .headers(headers -> headers.setBearerAuth(overridingToken))
        .retrieve()
        .bodyToMono(String.class)
        .block()
```

Kotlin

```
this.rest.get()
        .uri("https://other-service.example.com/endpoint")
        .headers{  headers -> headers.setBearerAuth(overridingToken)}
        .retrieve()
        .bodyToMono<String>()
        .block()
```

In this case, the filter will fall back and simply forward the request onto the rest of the web filter chain.

Unlike the OAuth 2.0 Client filter function, this filter function makes no attempt to renew the token, should it be expired. To obtain this level of support, please use the OAuth 2.0 Client filter.

`RestTemplate` support

There is no `RestTemplate` equivalent for `ServletBearerExchangeFilterFunction` at the moment, but you can propagate the request's bearer token quite simply with your own interceptor:

Java

```
@Bean
RestTemplate rest() {
    RestTemplate rest = new RestTemplate();
    rest.getInterceptors().add((request, body, execution) -> {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        if (authentication == null) {
            return execution.execute(request, body);
        }

        if (!(authentication.getCredentials() instanceof AbstractOAuth2Token)) {
            return execution.execute(request, body);
        }

        AbstractOAuth2Token token = (AbstractOAuth2Token)
authentication.getCredentials();
        request.getHeaders().setBearerAuth(token.getTokenValue());
        return execution.execute(request, body);
    });
    return rest;
}
```

Kotlin

```
@Bean
fun rest(): RestTemplate {
    val rest = RestTemplate()
    rest.interceptors.add(ClientHttpRequestInterceptor { request, body, execution
->
        val authentication: Authentication? =
SecurityContextHolder.getContext().authentication
        if (authentication != null) {
            execution.execute(request, body)
        }

        if (authentication!!.credentials !is AbstractOAuth2Token) {
            execution.execute(request, body)
        }

        val token: AbstractOAuth2Token = authentication.credentials as
AbstractOAuth2Token
        request.headers.setBearerAuth(token.tokenValue)
        execution.execute(request, body)
    })
    return rest
}
```

Unlike the OAuth 2.0 Authorized Client Manager, this filter interceptor makes no attempt to renew the token, should it be expired. To obtain this level of support, please create an interceptor using the OAuth 2.0 Authorized Client Manager.

12.3.26. Bearer Token Failure

A bearer token may be invalid for a number of reasons. For example, the token may no longer be active.

In these circumstances, Resource Server throws an `InvalidBearerTokenException`. Like other exceptions, this results in an OAuth 2.0 Bearer Token error response:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer error_code="invalid_token", error_description="Unsupported
algorithm of none", error_uri="https://tools.ietf.org/html/rfc6750#section-3.1"
```

Additionally, it is published as an `AuthenticationFailureBadCredentialsEvent`, which you can listen for in your application like so:

Java

```
@Component
public class FailureEvents {
    @EventListener
    public void onFailure(AuthenticationFailureBadCredentialsEvent badCredentials)
{
        if (badCredentials.getAuthentication() instanceof
BearerTokenAuthenticationToken) {
            // ... handle
        }
    }
}
```

Kotlin

```
@Component
class FailureEvents {
    @EventListener
    fun onFailure(badCredentials: AuthenticationFailureBadCredentialsEvent) {
        if (badCredentials.authentication is BearerTokenAuthenticationToken) {
            // ... handle
        }
    }
}
```

Chapter 13. SAML2

13.1. SAML 2.0 Login

The SAML 2.0 Login feature provides an application with the capability to act as a SAML 2.0 Relying Party, having users log in to the application by using their existing account at a SAML 2.0 Asserting Party (Okta, ADFS, etc).

SAML 2.0 Login is implemented by using the **Web Browser SSO Profile**, as specified in SAML 2 Profiles.

Since 2009, support for relying parties has existed as an extension project. In 2019, the process began to port that into Spring Security proper. This process is similar to the one started in 2017 for Spring Security's OAuth 2.0 support.

A working sample for SAML 2.0 Login is available in the Spring Security Samples repository.

Let's take a look at how SAML 2.0 Relying Party Authentication works within Spring Security. First, we see that, like OAuth 2.0 Login, Spring Security takes the user to a third-party for performing authentication. It does this through a series of redirects.

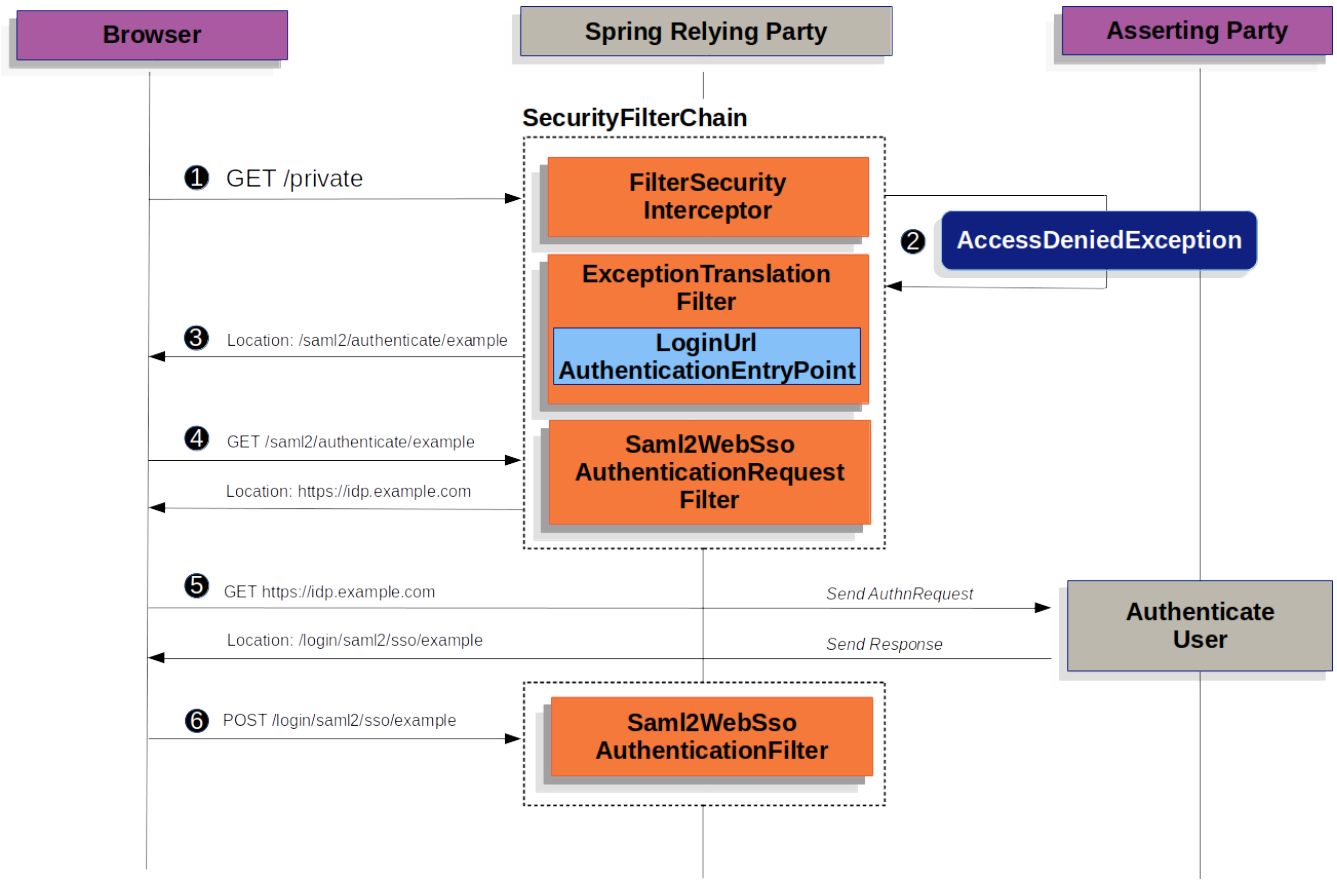


Figure 18. Redirecting to Asserting Party Authentication

The figure above builds off our `SecurityFilterChain` and `AbstractAuthenticationProcessingFilter` diagrams:

1. First, a user makes an unauthenticated request to the resource `/private` for which it is not authorized.

2. Spring Security's `FilterSecurityInterceptor` indicates that the unauthenticated request is *Denied* by throwing an `AccessDeniedException`.

3. Since the user lacks authorization, the `ExceptionTranslationFilter` initiates *Start Authentication*. The configured `AuthenticationEntryPoint` is an instance of `LoginUrlAuthenticationEntryPoint` which redirects to the `<saml2:AuthnRequest>` generating endpoint, `Saml2WebSsoAuthenticationRequestFilter`. Or, if you've configured more than one asserting party, it will first redirect to a picker page.

4. Next, the `Saml2WebSsoAuthenticationRequestFilter` creates, signs, serializes, and encodes a `<saml2:AuthnRequest>` using its configured `Saml2AuthenticationRequestFactory`.

5. Then, the browser takes this `<saml2:AuthnRequest>` and presents it to the asserting party. The asserting party attempts to authentication the user. If successful, it will return a `<saml2:Response>` back to the browser.

6. The browser then POSTs the `<saml2:Response>` to the assertion consumer service endpoint.

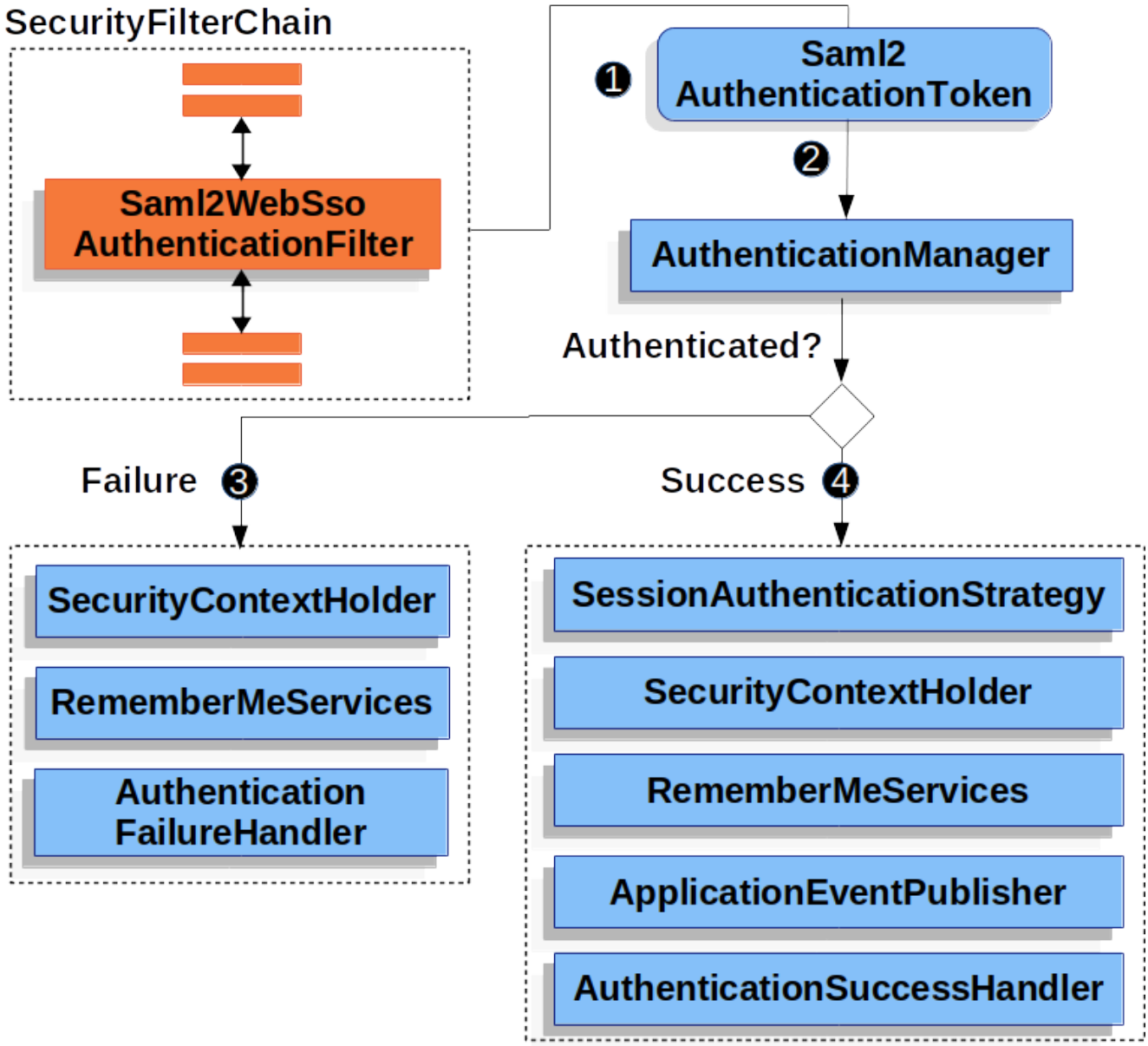


Figure 19. Authenticating a `<saml2:Response>`

The figure builds off our `SecurityFilterChain` diagram.

1 When the browser submits a `<saml2:Response>` to the application, it delegates to `Saml2WebSsoAuthenticationFilter`. This filter calls its configured `AuthenticationConverter` to create a `Saml2AuthenticationToken` by extracting the response from the `HttpServletRequest`. This converter additionally resolves the `RelyingPartyRegistration` and supplies it to `Saml2AuthenticationToken`.

2 Next, the filter passes the token to its configured `AuthenticationManager`. By default, it will use the `OpenSAML authentication provider`.

3 If authentication fails, then *Failure*

- The `SecurityContextHolder` is cleared out.
- The `AuthenticationEntryPoint` is invoked to restart the authentication process.

4 If authentication is successful, then *Success*.

- The `Authentication` is set on the `SecurityContextHolder`.

- The `Saml2WebSsoAuthenticationFilter` invokes `FilterChain#doFilter(request,response)` to continue with the rest of the application logic.

13.1.1. Minimal Dependencies

SAML 2.0 service provider support resides in `spring-security-saml2-service-provider`. It builds off of the OpenSAML library.

13.1.2. Minimal Configuration

When using Spring Boot, configuring an application as a service provider consists of two basic steps. First, include the needed dependencies and second, indicate the necessary asserting party metadata.

Also, this presupposes that you've already registered the relying party with your asserting party.

Specifying Identity Provider Metadata

In a Spring Boot application, to specify an identity provider's metadata, simply do:

```
spring:
  security:
    saml2:
      relyingparty:
        registration:
          adfs:
            identityprovider:
              entity-id: https://idp.example.com/issuer
              verification.credentials:
                - certificate-location: "classpath:idp.crt"
              singlesignon.url: https://idp.example.com/issuer/sso
              singlesignon.sign-request: false
```

where

- `https://idp.example.com/issuer` is the value contained in the `Issuer` attribute of the SAML responses that the identity provider will issue
- `classpath:idp.crt` is the location on the classpath for the identity provider's certificate for verifying SAML responses, and
- `https://idp.example.com/issuer/sso` is the endpoint where the identity provider is expecting `AuthnRequest` s.

And that's it!

Identity Provider and Asserting Party are synonymous, as are Service Provider and Relying Party. These are frequently abbreviated as AP and RP, respectively.

Runtime Expectations

As configured above, the application processes any `POST /login/saml2/sso/{registrationId}` request containing a `SAMLResponse` parameter:

```
POST /login/saml2/sso/adfs HTTP/1.1

SAMLResponse=PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZ...
```

There are two ways to see induce your asserting party to generate a `SAMLResponse`:

- First, you can navigate to your asserting party. It likely has some kind of link or button for each registered relying party that you can click to send the `SAMLResponse`.
- Second, you can navigate to a protected page in your app, for example, `http://localhost:8080`. Your app then redirects to the configured asserting party which then sends the `SAMLResponse`.

From here, consider jumping to:

- How SAML 2.0 Login Integrates with OpenSAML
- How to Use the `Saml2AuthenticatedPrincipal`
- How to Override or Replace Spring Boot's Auto Configuration

13.1.3. How SAML 2.0 Login Integrates with OpenSAML

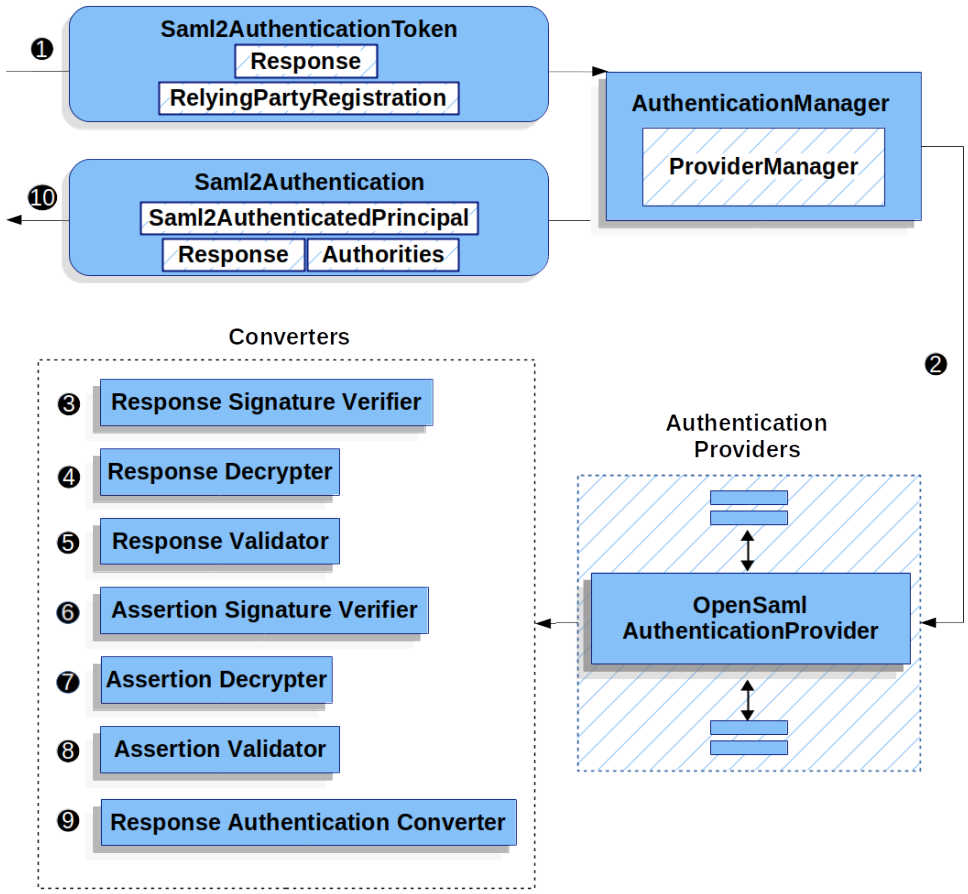
Spring Security's SAML 2.0 support has a couple of design goals:

- First, rely on a library for SAML 2.0 operations and domain objects. To achieve this, Spring Security uses OpenSAML.
- Second, ensure this library is not required when using Spring Security's SAML support. To achieve this, any interfaces or classes where Spring Security uses OpenSAML in the contract remain encapsulated. This makes it possible for you to switch out OpenSAML for some other library or even an unsupported version of OpenSAML.

As a natural outcome of the above two goals, Spring Security's SAML API is quite small relative to other modules. Instead, classes like `OpenSaml4AuthenticationRequestFactory` and `OpenSaml4AuthenticationProvider` expose `Converter` s that customize various steps in the authentication process.

For example, once your application receives a `SAMLResponse` and delegates to `Saml2WebSsoAuthenticationFilter`, the filter will delegate to `OpenSaml4AuthenticationProvider`.

For backward compatibility, Spring Security will use the latest OpenSAML 3 by default. Note, though that OpenSAML 3 has reached it's end-of-life and updating to OpenSAML 4.x is recommended. For that reason, Spring Security supports both OpenSAML 3.x and 4.x. If you manage your OpenSAML dependency to 4.x, then Spring Security will select its OpenSAML 4.x implementations.



This figure builds off of the `Saml2WebSsoAuthenticationFilter` diagram.

1. The `Saml2WebSsoAuthenticationFilter` formulates the `Saml2AuthenticationToken` and invokes the `AuthenticationManager`.

2. The `AuthenticationManager` invokes the OpenSAML authentication provider.

3. The authentication provider deserializes the response into an OpenSAML `Response` and checks its signature. If the signature is invalid, authentication fails.

4. Then, the provider decrypts any `EncryptedAssertion` elements. If any decryptions fail, authentication fails.

5. Next, the provider validates the response's `Issuer` and `Destination` values. If they don't match what's in the `RelyingPartyRegistration`, authentication fails.

❻ After that, the provider verifies the signature of each `Assertion`. If any signature is invalid, authentication fails. Also, if neither the response nor the assertions have signatures, authentication fails. Either the response or all the assertions must have signatures.

❼ Then, the provider decrypts any `EncryptedID` or `EncryptedAttribute` elements. If any decryptions fail, authentication fails.

❽ Next, the provider validates each assertion's `ExpiresAt` and `NotBefore` timestamps, the `<Subject>` and any `<AudienceRestriction>` conditions. If any validations fail, authentication fails.

❾ Following that, the provider takes the first assertion's `AttributeStatement` and maps it to a `Map<String, List<Object>>`. It also grants the `ROLE_USER` granted authority.

❿ And finally, it takes the `NameID` from the first assertion, the `Map` of attributes, and the `GrantedAuthority` and constructs a `Saml2AuthenticatedPrincipal`. Then, it places that principal and the authorities into a `Saml2Authentication`.

The resulting `Authentication#getPrincipal` is a Spring Security `Saml2AuthenticatedPrincipal` object, and `Authentication#getName` maps to the first assertion's `NameID` element.

Customizing OpenSAML Configuration

Any class that uses both Spring Security and OpenSAML should statically initialize `OpenSamlInitializationService` at the beginning of the class, like so:

Java

```
static {
    OpenSamlInitializationService.initialize();
}
```

Kotlin

```
companion object {
    init {
        OpenSamlInitializationService.initialize()
    }
}
```

This replaces OpenSAML's `InitializationService#initialize`.

Occasionally, it can be valuable to customize how OpenSAML builds, marshalls, and unmarshalls SAML objects. In these circumstances, you may instead want to call `OpenSamlInitializationService#requireInitialize(Consumer)` that gives you access to OpenSAML's `XMLObjectProviderFactory`.

For example, when sending an unsigned AuthNRequest, you may want to force reauthentication. In that case, you can register your own `AuthnRequestMarshaller`, like so:

Java

```
static {
    OpenSamlInitializationService.requireInitialize(factory -> {
        AuthnRequestMarshaller marshaller = new AuthnRequestMarshaller() {
            @Override
            public Element marshall(XMLObject object, Element element) throws
MarshallingException {
                configureAuthnRequest((AuthnRequest) object);
                return super.marshall(object, element);
            }

            public Element marshall(XMLObject object, Document document) throws
MarshallingException {
                configureAuthnRequest((AuthnRequest) object);
                return super.marshall(object, document);
            }

            private void configureAuthnRequest(AuthnRequest authnRequest) {
                authnRequest.setForceAuthn(true);
            }
        }


factory.getMarshallerFactory().registerMarshaller(AuthnRequest.DEFAULT_ELEMENT_NAM
E, marshaller);
    });
}
```

Kotlin

```
companion object {
    init {
        OpenSamlInitializationService.requireInitialize {
            val marshaller = object : AuthnRequestMarshaller() {
                override fun marshall(xmlObject: XMLObject, element: Element):
Element {
                    configureAuthnRequest(xmlObject as AuthnRequest)
                    return super.marshall(xmlObject, element)
                }

                override fun marshall(xmlObject: XMLObject, document: Document):
Element {
                    configureAuthnRequest(xmlObject as AuthnRequest)
                    return super.marshall(xmlObject, document)
                }

                private fun configureAuthnRequest(authnRequest: AuthnRequest) {
                    authnRequest.isForceAuthn = true
                }
            }

it.marshallerFactory.registerMarshaller(AuthnRequest.DEFAULT_ELEMENT_NAME,
marshaller)
        }
    }
}
```

The `requireInitialize` method may only be called once per application instance.

13.1.4. Overriding or Replacing Boot Auto Configuration

There are two `@Bean` s that Spring Boot generates for a relying party.

The first is a `WebSecurityConfigurerAdapter` that configures the app as a relying party. When including `spring-security-saml2-service-provider`, the `WebSecurityConfigurerAdapter` looks like:

Example 141. Default JWT Configuration

Java

```
protected void configure(HttpSecurity http) {
    http
        .authorizeRequests(authorize -> authorize
            .anyRequest().authenticated()
        )
        .saml2Login(withDefaults());
}
```

Kotlin

```
fun configure(http: HttpSecurity) {
    http {
        authorizeRequests {
            authorize(anyRequest, authenticated)
        }
        saml2Login { }
    }
}
```

If the application doesn't expose a `WebSecurityConfigurerAdapter` bean, then Spring Boot will expose the above default one.

You can replace this by exposing the bean within the application:

Example 142. Custom SAML 2.0 Login Configuration

Java

```
@EnableWebSecurity
public class MyCustomSecurityConfiguration extends WebSecurityConfigurerAdapter {
    protected void configure(HttpSecurity http) {
        http
            .authorizeRequests(authorize -> authorize
                .mvcMatchers("/messages/**").hasAuthority("ROLE_USER")
                .anyRequest().authenticated()
            )
            .saml2Login(withDefaults());
    }
}
```

Kotlin

```
@EnableWebSecurity
class MyCustomSecurityConfiguration : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            authorizeRequests {
                authorize("/messages/**", hasAuthority("ROLE_USER"))
                authorize(anyRequest, authenticated)
            }
            saml2Login {
            }
        }
    }
}
```

The above requires the role of `USER` for any URL that starts with `/messages/`.

The second `@Bean` Spring Boot creates is a `RelyingPartyRegistrationRepository`, which represents the asserting party and relying party metadata. This includes things like the location of the SSO endpoint the relying party should use when requesting authentication from the asserting party.

You can override the default by publishing your own `RelyingPartyRegistrationRepository` bean. For example, you can look up the asserting party's configuration by hitting its metadata endpoint like so:

Example 143. Relying Party Registration Repository

Java

```
@Value("${metadata.location}")
String assertingPartyMetadataLocation;

@Bean
public RelyingPartyRegistrationRepository relyingPartyRegistrations() {
    RelyingPartyRegistration registration = RelyingPartyRegistrations
            .fromMetadataLocation(assertingPartyMetadataLocation)
            .registrationId("example")
            .build();
    return new InMemoryRelyingPartyRegistrationRepository(registration);
}
```

Kotlin

```
@Value("\${metadata.location}")
var assertingPartyMetadataLocation: String? = null

@Bean
open fun relyingPartyRegistrations(): RelyingPartyRegistrationRepository? {
    val registration = RelyingPartyRegistrations
        .fromMetadataLocation(assertingPartyMetadataLocation)
        .registrationId("example")
        .build()
    return InMemoryRelyingPartyRegistrationRepository(registration)
}
```

Or you can provide each detail manually, as you can see below:

Example 144. Relying Party Registration Repository Manual Configuration

Java

```
@Value("${verification.key}")
File verificationKey;

@Bean
public RelyingPartyRegistrationRepository relyingPartyRegistrations() throws
Exception {
    X509Certificate certificate =
X509Support.decodeCertificate(this.verificationKey);
    Saml2X509Credential credential =
Saml2X509Credential.verification(certificate);
    RelyingPartyRegistration registration = RelyingPartyRegistration
            .withRegistrationId("example")
            .assertingPartyDetails(party -> party
                .entityId("https://idp.example.com/issuer")
                .singleSignOnServiceLocation("https://idp.example.com/SSO.saml2")
                .wantAuthnRequestsSigned(false)
                .verificationX509Credentials(c -> c.add(credential))
            )
            .build();
    return new InMemoryRelyingPartyRegistrationRepository(registration);
}
```

Kotlin

```
@Value("\${verification.key}")
var verificationKey: File? = null

@Bean
open fun relyingPartyRegistrations(): RelyingPartyRegistrationRepository {
    val certificate: X509Certificate? =
X509Support.decodeCertificate(verificationKey!!)
    val credential: Saml2X509Credential =
Saml2X509Credential.verification(certificate)
    val registration = RelyingPartyRegistration
        .withRegistrationId("example")
        .assertingPartyDetails { party: AssertingPartyDetails.Builder ->
            party
                .entityId("https://idp.example.com/issuer")
                .singleSignOnServiceLocation("https://idp.example.com/SSO.saml2")
                .wantAuthnRequestsSigned(false)
                .verificationX509Credentials { c:
MutableCollection<Saml2X509Credential?> ->
                    c.add(
                        credential
                    )
                }
        }
        .build()
    return InMemoryRelyingPartyRegistrationRepository(registration)
}
```

Note that `X509Support` is an OpenSAML class, used here in the snippet for brevity

Alternatively, you can directly wire up the repository using the DSL, which will also override the auto-configured `WebSecurityConfigurerAdapter`:

Example 145. Custom Relying Party Registration DSL

Java

```
@EnableWebSecurity
public class MyCustomSecurityConfiguration extends WebSecurityConfigurerAdapter {
    protected void configure(HttpSecurity http) {
        http
            .authorizeRequests(authorize -> authorize
                .mvcMatchers("/messages/**").hasAuthority("ROLE_USER")
                .anyRequest().authenticated()
            )
            .saml2Login(saml2 -> saml2
                .relyingPartyRegistrationRepository(relyingPartyRegistrations())
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class MyCustomSecurityConfiguration : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            authorizeRequests {
                authorize("/messages/**", hasAuthority("ROLE_USER"))
                authorize(anyRequest, authenticated)
            }
            saml2Login {
                relyingPartyRegistrationRepository = relyingPartyRegistrations()
            }
        }
    }
}
```

A relying party can be multi-tenant by registering more than one relying party in the `RelyingPartyRegistrationRepository`.

13.1.5. RelyingPartyRegistration

A `RelyingPartyRegistration` instance represents a link between an relying party and assering party's metadata.

In a `RelyingPartyRegistration`, you can provide relying party metadata like its `Issuer` value, where it expects SAML Responses to be sent to, and any credentials that it owns for the purposes of signing or decrypting payloads.

Also, you can provide asserting party metadata like its `Issuer` value, where it expects

AuthnRequests to be sent to, and any public credentials that it owns for the purposes of the relying party verifying or encrypting payloads.

The following `RelyingPartyRegistration` is the minimum required for most setups:

Java

```
RelyingPartyRegistration relyingPartyRegistration = RelyingPartyRegistrations
        .fromMetadataLocation("https://ap.example.org/metadata")
        .registrationId("my-id")
        .build();
```

Kotlin

```
val relyingPartyRegistration = RelyingPartyRegistrations
    .fromMetadataLocation("https://ap.example.org/metadata")
    .registrationId("my-id")
    .build()
```

Note that you can also create a `RelyingPartyRegistration` from an arbitrary `InputStream` source. One such example is when the metadata is stored in a database:

```
String xml = fromDatabase();
try (InputStream source = new ByteArrayInputStream(xml.getBytes())) {
    RelyingPartyRegistration relyingPartyRegistration = RelyingPartyRegistrations
            .fromMetadata(source)
            .registrationId("my-id")
            .build();
}
```

Though a more sophisticated setup is also possible, like so:

Java

```
RelyingPartyRegistration relyingPartyRegistration =
RelyingPartyRegistration.withRegistrationId("my-id")
        .entityId("{baseUrl}/{registrationId}")
        .decryptionX509Credentials(c -> c.add(relyingPartyDecryptingCredential()))
        .assertionConsumerServiceLocation("/my-login-endpoint/{registrationId}")
        .assertingPartyDetails(party -> party
                .entityId("https://ap.example.org")
                .verificationX509Credentials(c ->
c.add(assertingPartyVerifyingCredential()))
                .singleSignOnServiceLocation("https://ap.example.org/SSO.saml2")
        )
        .build();
```

Kotlin

```
val relyingPartyRegistration =
    RelyingPartyRegistration.withRegistrationId("my-id")
        .entityId("{baseUrl}/{registrationId}")
        .decryptionX509Credentials { c: MutableCollection<Saml2X509Credential?> ->
            c.add(relyingPartyDecryptingCredential())
        }
        .assertionConsumerServiceLocation("/my-login-endpoint/{registrationId}")
        .assertingPartyDetails { party -> party
                .entityId("https://ap.example.org")
                .verificationX509Credentials { c ->
c.add(assertingPartyVerifyingCredential()) }
                .singleSignOnServiceLocation("https://ap.example.org/SSO.saml2")
        }
        .build()
```

The top-level metadata methods are details about the relying party. The methods inside `assertingPartyDetails` are details about the asserting party.

The location where a relying party is expecting SAML Responses is the Assertion Consumer Service Location.

The default for the relying party's `entityId` is `{baseUrl}/saml2/service-provider-metadata/{registrationId}`. This is this value needed when configuring the asserting party to know about your relying party.

The default for the `assertionConsumerServiceLocation` is `/login/saml2/sso/{registrationId}`. It's mapped by default to `Saml2WebSsoAuthenticationFilter` in the filter chain.

URI Patterns

You probably noticed in the above examples the `{baseUrl}` and `{registrationId}` placeholders.

These are useful for generating URIs. As such, the relying party's `entityId` and `assertionConsumerServiceLocation` support the following placeholders:

- `baseUrl` - the scheme, host, and port of a deployed application
- `registrationId` - the registration id for this relying party
- `baseScheme` - the scheme of a deployed application
- `baseHost` - the host of a deployed application
- `basePort` - the port of a deployed application

For example, the `assertionConsumerServiceLocation` defined above was:

```
/my-login-endpoint/{registrationId}
```

which in a deployed application would translate to

```
/my-login-endpoint/adfs
```

The `entityId` above was defined as:

```
{baseUrl}/{registrationId}
```

which in a deployed application would translate to

```
https://rp.example.com/adfs
```

Credentials

You also likely noticed the credential that was used.

Oftentimes, a relying party will use the same key to sign payloads as well as decrypt them. Or it will use the same key to verify payloads as well as encrypt them.

Because of this, Spring Security ships with `Saml2X509Credential`, a SAML-specific credential that simplifies configuring the same key for different use cases.

At a minimum, it's necessary to have a certificate from the asserting party so that the asserting party's signed responses can be verified.

To construct a `Saml2X509Credential` that you'll use to verify assertions from the asserting party, you can load the file and use the `CertificateFactory` like so:

Java

```
Resource resource = new ClassPathResource("ap.crt");
try (InputStream is = resource.getInputStream()) {
    X509Certificate certificate = (X509Certificate)
            CertificateFactory.getInstance("X.509").generateCertificate(is);
    return Saml2X509Credential.verification(certificate);
}
```

Kotlin

```
val resource = ClassPathResource("ap.crt")
resource.inputStream.use {
    return Saml2X509Credential.verification(
        CertificateFactory.getInstance("X.509").generateCertificate(it) as
X509Certificate?
    )
}
```

Let's say that the asserting party is going to also encrypt the assertion. In that case, the relying party will need a private key to be able to decrypt the encrypted value.

In that case, you'll need an `RSAPrivateKey` as well as its corresponding `X509Certificate`. You can load the first using Spring Security's `RsaKeyConverters` utility class and the second as you did before:

Java

```
X509Certificate certificate = relyingPartyDecryptionCertificate();
Resource resource = new ClassPathResource("rp.crt");
try (InputStream is = resource.getInputStream()) {
    RSAPrivateKey rsa = RsaKeyConverters.pkcs8().convert(is);
    return Saml2X509Credential.decryption(rsa, certificate);
}
```

Kotlin

```
val certificate: X509Certificate = relyingPartyDecryptionCertificate()
val resource = ClassPathResource("rp.crt")
resource.inputStream.use {
    val rsa: RSAPrivateKey = RsaKeyConverters.pkcs8().convert(it)
    return Saml2X509Credential.decryption(rsa, certificate)
}
```

When you specify the locations of these files as the appropriate Spring Boot properties, then Spring Boot will perform these conversions for you.

Resolving the Relying Party from the Request

As seen so far, Spring Security resolves the `RelyingPartyRegistration` by looking for the registration id in the URI path.

There are a number of reasons you may want to customize. Among them:

- You may know that you will never be a multi-tenant application and so want to have a simpler URL scheme
- You may identify tenants in a way other than by the URI path

To customize the way that a `RelyingPartyRegistration` is resolved, you can configure a custom `Converter<HttpServletRequest, RelyingPartyRegistration>`. The default looks up the registration id from the URI's last path element and looks it up in your `RelyingPartyRegistrationRepository`.

You can provide a simpler resolver that, for example, always returns the same relying party:

Java

```
public class SingleRelyingPartyRegistrationResolver
        implements Converter<HttpServletRequest, RelyingPartyRegistration> {

    @Override
    public RelyingPartyRegistration convert(HttpServletRequest request) {
        return this.relyingParty;
    }
}
```

Kotlin

```
class SingleRelyingPartyRegistrationResolver : Converter<HttpServletRequest?,
RelyingPartyRegistration?> {
    override fun convert(request: HttpServletRequest?): RelyingPartyRegistration?
{
        return this.relyingParty
    }
}
```

Then, you can provide this resolver to the appropriate filters that produce `<saml2:AuthnRequest>` s, authenticate `<saml2:Response>` s, and produce `<saml2:SPSSODescriptor>` metadata.

Remember that if you have any placeholders in your `RelyingPartyRegistration`, your resolver implementation should resolve them.

Duplicated Relying Party Configurations

When an application uses multiple asserting parties, some configuration is duplicated between `RelyingPartyRegistration` instances:

- The relying party's `entityId`
- Its `assertionConsumerServiceLocation`, and
- Its credentials, for example its signing or decryption credentials

What's nice about this setup is credentials may be more easily rotated for some identity providers vs others.

The duplication can be alleviated in a few different ways.

First, in YAML this can be alleviated with references, like so:

```
spring:
  security:
    saml2:
      relyingparty:
        okta:
          signing.credentials: &relying-party-credentials
            - private-key-location: classpath:rp.key
            - certificate-location: classpath:rp.crt
          identityprovider:
            entity-id: ...
        azure:
          signing.credentials: *relying-party-credentials
          identityprovider:
            entity-id: ...
```

Second, in a database, it's not necessary to replicate `RelyingPartyRegistration` 's model.

Third, in Java, you can create a custom configuration method, like so:

Java

```
private RelyingPartyRegistration.Builder
        addRelyingPartyDetails(RelyingPartyRegistration.Builder builder) {

    Saml2X509Credential signingCredential = ...
    builder.signingX509Credentials(c -> c.addAll(signingCredential));
    // ... other relying party configurations
}

@Bean
public RelyingPartyRegistrationRepository relyingPartyRegistrations() {
    RelyingPartyRegistration okta = addRelyingPartyDetails(
            RelyingPartyRegistrations
                .fromMetadataLocation(oktaMetadataUrl)
                .registrationId("okta")).build();

    RelyingPartyRegistration azure = addRelyingPartyDetails(
            RelyingPartyRegistrations
                .fromMetadataLocation(oktaMetadataUrl)
                .registrationId("azure")).build();

    return new InMemoryRelyingPartyRegistrationRepository(okta, azure);
}
```

Kotlin

```
private fun addRelyingPartyDetails(builder: RelyingPartyRegistration.Builder):
RelyingPartyRegistration.Builder {
    val signingCredential: Saml2X509Credential = ...
    builder.signingX509Credentials { c: MutableCollection<Saml2X509Credential?> ->
        c.add(
            signingCredential
        )
    }
    // ... other relying party configurations
}

@Bean
open fun relyingPartyRegistrations(): RelyingPartyRegistrationRepository? {
    val okta = addRelyingPartyDetails(
        RelyingPartyRegistrations
            .fromMetadataLocation(oktaMetadataUrl)
            .registrationId("okta")
    ).build()
    val azure = addRelyingPartyDetails(
        RelyingPartyRegistrations
            .fromMetadataLocation(oktaMetadataUrl)
            .registrationId("azure")
    ).build()
    return InMemoryRelyingPartyRegistrationRepository(okta, azure)
}
```

13.1.6. Producing `<saml2:AuthnRequest>` s

As stated earlier, Spring Security's SAML 2.0 support produces a `<saml2:AuthnRequest>` to commence authentication with the asserting party.

Spring Security achieves this in part by registering the `Saml2WebSsoAuthenticationRequestFilter` in the filter chain. This filter by default responds to endpoint `/saml2/authenticate/{registrationId}`.

For example, if you were deployed to `https://rp.example.com` and you gave your registration an ID of `okta`, you could navigate to:

`https://rp.example.org/saml2/authenticate/ping`

and the result would be a redirect that included a `SAMLRequest` parameter containing the signed, deflated, and encoded `<saml2:AuthnRequest>`.

Changing How the `<saml2:AuthnRequest>` Gets Sent

By default, Spring Security signs each `<saml2:AuthnRequest>` and send it as a GET to the asserting party.

Many asserting parties don't require a signed `<saml2:AuthnRequest>`. This can be configured

automatically via `RelyingPartyRegistrations`, or you can supply it manually, like so:

Example 146. Not Requiring Signed AuthnRequests

Boot

```
spring:
  security:
    saml2:
      relyingparty:
        okta:
          identityprovider:
            entity-id: ...
            singlesignon.sign-request: false
```

Java

```
RelyingPartyRegistration relyingPartyRegistration =
RelyingPartyRegistration.withRegistrationId("okta")
        // ...
        .assertingPartyDetails(party -> party
            // ...
            .wantAuthnRequestsSigned(false)
        )
        .build();
```

Kotlin

```
var relyingPartyRegistration: RelyingPartyRegistration =
    RelyingPartyRegistration.withRegistrationId("okta")
        // ...
        .assertingPartyDetails { party: AssertingPartyDetails.Builder -> party
                // ...
                .wantAuthnRequestsSigned(false)
        }
        .build();
```

Otherwise, you will need to specify a private key to `RelyingPartyRegistration#signingX509Credentials` so that Spring Security can sign the `<saml2:AuthnRequest>` before sending.

By default, Spring Security will sign the `<saml2:AuthnRequest>` using `rsa-sha256`, though some asserting parties will require a different algorithm, as indicated in their metadata.

You can configure the algorithm based on the asserting party's metadata using `RelyingPartyRegistrations`.

Or, you can provide it manually:

Java

```
String metadataLocation = "classpath:asserting-party-metadata.xml";
RelyingPartyRegistration relyingPartyRegistration =
RelyingPartyRegistrations.fromMetadataLocation(metadataLocation)
        // ...
        .assertingPartyDetails((party) -> party
            // ...
            .signingAlgorithms((sign) ->
sign.add(SignatureConstants.ALGO_ID_SIGNATURE_RSA_SHA512))
        )
        .build();
```

Kotlin

```
var metadataLocation = "classpath:asserting-party-metadata.xml"
var relyingPartyRegistration: RelyingPartyRegistration =
    RelyingPartyRegistrations.fromMetadataLocation(metadataLocation)
        // ...
        .assertingPartyDetails { party: AssertingPartyDetails.Builder -> party
                // ...
                .signingAlgorithms { sign: MutableList<String?> ->
                    sign.add(
                        SignatureConstants.ALGO_ID_SIGNATURE_RSA_SHA512
                    )
                }
        }
        .build();
```

The snippet above uses the OpenSAML `SignatureConstants` class to supply the algorithm name. But, that's just for convenience. Since the datatype is `String`, you can supply the name of the algorithm directly.

Some asserting parties require that the `<saml2:AuthnRequest>` be POSTed. This can be configured automatically via `RelyingPartyRegistrations`, or you can supply it manually, like so:

Java

```
RelyingPartyRegistration relyingPartyRegistration =
RelyingPartyRegistration.withRegistrationId("okta")
        // ...
        .assertingPartyDetails(party -> party
            // ...
            .singleSignOnServiceBinding(Saml2MessageBinding.POST)
        )
        .build();
```

Kotlin

```
var relyingPartyRegistration: RelyingPartyRegistration? =
    RelyingPartyRegistration.withRegistrationId("okta")
        // ...
        .assertingPartyDetails { party: AssertingPartyDetails.Builder -> party
            // ...
            .singleSignOnServiceBinding(Saml2MessageBinding.POST)
        }
        .build()
```

Customizing OpenSAML's `AuthnRequest` Instance

There are a number of reasons that you may want to adjust an `AuthnRequest`. For example, you may want `ForceAuthN` to be set to `true`, which Spring Security sets to `false` by default.

If you don't need information from the `HttpServletRequest` to make your decision, then the easiest way is to register a custom `AuthnRequestMarshaller` with OpenSAML. This will give you access to post-process the `AuthnRequest` instance before it's serialized.

But, if you do need something from the request, then you can use create a custom `Saml2AuthenticationRequestContext` implementation and then a `Converter<Saml2AuthenticationRequestContext, AuthnRequest>` to build an `AuthnRequest` yourself, like so:

Java

```
@Component
public class AuthnRequestConverter implements
        Converter<MySaml2AuthenticationRequestContext, AuthnRequest> {

    private final AuthnRequestBuilder authnRequestBuilder;
    private final IssuerBuilder issuerBuilder;

    // ... constructor

    public AuthnRequest convert(Saml2AuthenticationRequestContext context) {
        MySaml2AuthenticationRequestContext myContext =
(MySaml2AuthenticationRequestContext) context;
        Issuer issuer = issuerBuilder.buildObject();
        issuer.setValue(myContext.getIssuer());

        AuthnRequest authnRequest = authnRequestBuilder.buildObject();
        authnRequest.setIssuer(issuer);
        authnRequest.setDestination(myContext.getDestination());

authnRequest.setAssertionConsumerServiceURL(myContext.getAssertionConsumerServiceU
rl());

        // ... additional settings

        authRequest.setForceAuthn(myContext.getForceAuthn());
        return authnRequest;
    }
}
```

Kotlin

```
@Component
class AuthnRequestConverter : Converter<MySaml2AuthenticationRequestContext,
AuthnRequest> {
    private val authnRequestBuilder: AuthnRequestBuilder? = null
    private val issuerBuilder: IssuerBuilder? = null

    // ... constructor
    override fun convert(context: MySaml2AuthenticationRequestContext):
AuthnRequest {
        val myContext: MySaml2AuthenticationRequestContext = context
        val issuer: Issuer = issuerBuilder.buildObject()
        issuer.value = myContext.getIssuer()
        val authnRequest: AuthnRequest = authnRequestBuilder.buildObject()
        authnRequest.issuer = issuer
        authnRequest.destination = myContext.getDestination()
        authnRequest.assertionConsumerServiceURL =
myContext.getAssertionConsumerServiceUrl()

        // ... additional settings
        authRequest.setForceAuthn(myContext.getForceAuthn())
        return authnRequest
    }
}
```

Then, you can construct your own `Saml2AuthenticationRequestContextResolver` and `Saml2AuthenticationRequestFactory` and publish them as `@Bean` s:

Java

```
@Bean
Saml2AuthenticationRequestContextResolver authenticationRequestContextResolver() {
    Saml2AuthenticationRequestContextResolver resolver =
            new DefaultSaml2AuthenticationRequestContextResolver();
    return request -> {
        Saml2AuthenticationRequestContext context = resolver.resolve(request);
        return new MySaml2AuthenticationRequestContext(context,
request.getParameter("force") != null);
    };
}

@Bean
Saml2AuthenticationRequestFactory authenticationRequestFactory(
        AuthnRequestConverter authnRequestConverter) {

    OpenSaml4AuthenticationRequestFactory authenticationRequestFactory =
            new OpenSaml4AuthenticationRequestFactory();

authenticationRequestFactory.setAuthenticationRequestContextConverter(authnRequest
Converter);
    return authenticationRequestFactory;
}
```

Kotlin

```
@Bean
open fun authenticationRequestContextResolver():
Saml2AuthenticationRequestContextResolver {
    val resolver: Saml2AuthenticationRequestContextResolver =
DefaultSaml2AuthenticationRequestContextResolver()
    return Saml2AuthenticationRequestContextResolver { request: HttpServletRequest
->
        val context = resolver.resolve(request)
        MySaml2AuthenticationRequestContext(
            context,
            request.getParameter("force") != null
        )
    }
}

@Bean
open fun authenticationRequestFactory(
    authnRequestConverter: AuthnRequestConverter?
): Saml2AuthenticationRequestFactory? {
    val authenticationRequestFactory = OpenSaml4AuthenticationRequestFactory()

authenticationRequestFactory.setAuthenticationRequestContextConverter(authnRequest
Converter)
    return authenticationRequestFactory
}
```

13.1.7. Authenticating `<saml2:Response>` s

To verify SAML 2.0 Responses, Spring Security uses `OpenSaml4AuthenticationProvider` by default.

You can configure this in a number of ways including:

1. Setting a clock skew to timestamp validation
2. Mapping the response to a list of `GrantedAuthority` instances
3. Customizing the strategy for validating assertions
4. Customizing the strategy for decrypting response and assertion elements

To configure these, you'll use the `saml2Login#authenticationManager` method in the DSL.

Setting a Clock Skew

It's not uncommon for the asserting and relying parties to have system clocks that aren't perfectly synchronized. For that reason, you can configure `OpenSaml4AuthenticationProvider` 's default assertion validator with some tolerance:

Java

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        OpenSaml4AuthenticationProvider authenticationProvider = new
OpenSaml4AuthenticationProvider();

authenticationProvider.setAssertionValidator(OpenSaml4AuthenticationProvider
                .createDefaultAssertionValidator(assertionToken -> {
                    Map<String, Object> params = new HashMap<>();
                    params.put(CLOCK_SKEW, Duration.ofMinutes(10).toMillis());
                    // ... other validation parameters
                    return new ValidationContext(params);
                })
        );

        http
            .authorizeRequests(authz -> authz
                .anyRequest().authenticated()
            )
            .saml2Login(saml2 -> saml2
                .authenticationManager(new
ProviderManager(authenticationProvider))
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
open class SecurityConfig : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        val authenticationProvider = OpenSaml4AuthenticationProvider()
        authenticationProvider.setAssertionValidator(
            OpenSaml4AuthenticationProvider

.createDefaultAssertionValidator(Converter<OpenSaml4AuthenticationProvider.Asserti
onToken, ValidationContext> {
                    val params: MutableMap<String, Any> = HashMap()
                    params[CLOCK_SKEW] =
                        Duration.ofMinutes(10).toMillis()
                    ValidationContext(params)
                })
        )
        http {
            authorizeRequests {
                authorize(anyRequest, authenticated)
            }
            saml2Login {
                authenticationManager = ProviderManager(authenticationProvider)
            }
        }
    }
}
```

Coordinating with a `UserDetailsService`

Or, perhaps you would like to include user details from a legacy `UserDetailsService`. In that case, the response authentication converter can come in handy, as can be seen below:

Java

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    UserDetailsService userDetailsService;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        OpenSaml4AuthenticationProvider authenticationProvider = new
OpenSaml4AuthenticationProvider();
        authenticationProvider.setResponseAuthenticationConverter(responseToken ->
{
            Saml2Authentication authentication = OpenSaml4AuthenticationProvider
                    .createDefaultResponseAuthenticationConverter() ①
                    .convert(responseToken);
            Assertion assertion =
responseToken.getResponse().getAssertions().get(0);
            String username = assertion.getSubject().getNameID().getValue();
            UserDetails userDetails =
this.userDetailsService.loadUserByUsername(username); ②
            return MySaml2Authentication(userDetails, authentication); ③
        });

        http
            .authorizeRequests(authz -> authz
                .anyRequest().authenticated()
            )
            .saml2Login(saml2 -> saml2
                .authenticationManager(new
ProviderManager(authenticationProvider))
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
open class SecurityConfig : WebSecurityConfigurerAdapter() {
    @Autowired
    var userDetailsService: UserDetailsService? = null

    override fun configure(http: HttpSecurity) {
        val authenticationProvider = OpenSaml4AuthenticationProvider()
        authenticationProvider.setResponseAuthenticationConverter { responseToken:
OpenSaml4AuthenticationProvider.ResponseToken ->
            val authentication = OpenSaml4AuthenticationProvider
                .createDefaultResponseAuthenticationConverter() ①
                .convert(responseToken)
            val assertion: Assertion = responseToken.response.assertions[0]
            val username: String = assertion.subject.nameID.value
            val userDetails = userDetailsService!!.loadUserByUsername(username) ②
            MySaml2Authentication(userDetails, authentication) ③
        }
        http {
            authorizeRequests {
                authorize(anyRequest, authenticated)
            }
            saml2Login {
                authenticationManager = ProviderManager(authenticationProvider)
            }
        }
    }
}
```

① First, call the default converter, which extracts attributes and authorities from the response

② Second, call the `UserDetailsService` using the relevant information

③ Third, return a custom authentication that includes the user details

It's not required to call `OpenSaml4AuthenticationProvider` 's default authentication converter. It returns a `Saml2AuthenticatedPrincipal` containing the attributes it extracted from `AttributeStatement` s as well as the single `ROLE_USER` authority.

Performing Additional Response Validation

`OpenSaml4AuthenticationProvider` validates the `Issuer` and `Destination` values right after decrypting the `Response`. You can customize the validation by extending the default validator concatenating with your own response validator, or you can replace it entirely with yours.

For example, you can throw a custom exception with any additional information available in the `Response` object, like so:

```
OpenSaml4AuthenticationProvider provider = new OpenSaml4AuthenticationProvider();
provider.setResponseValidator((responseToken) -> {
    Saml2ResponseValidatorResult result = OpenSamlAuthenticationProvider
        .createDefaultResponseValidator()
        .convert(responseToken)
        .concat(myCustomValidator.convert(responseToken));
    if (!result.getErrors().isEmpty()) {
        String inResponseTo = responseToken.getInResponseTo();
        throw new CustomSaml2AuthenticationException(result, inResponseTo);
    }
    return result;
});
```

Performing Additional Assertion Validation

`OpenSaml4AuthenticationProvider` performs minimal validation on SAML 2.0 Assertions. After verifying the signature, it will:

1. Validate `<AudienceRestriction>` and `<DelegationRestriction>` conditions
2. Validate `<SubjectConfirmation>` s, expect for any IP address information

To perform additional validation, you can configure your own assertion validator that delegates to `OpenSaml4AuthenticationProvider` 's default and then performs its own.

For example, you can use OpenSAML's `OneTimeUseConditionValidator` to also validate a `<OneTimeUse>` condition, like so:

Java

```
OpenSaml4AuthenticationProvider provider = new OpenSaml4AuthenticationProvider();
OneTimeUseConditionValidator validator = ...;
provider.setAssertionValidator(assertionToken -> {
    Saml2ResponseValidatorResult result = OpenSaml4AuthenticationProvider
            .createDefaultAssertionValidator()
            .convert(assertionToken);
    Assertion assertion = assertionToken.getAssertion();
    OneTimeUse oneTimeUse = assertion.getConditions().getOneTimeUse();
    ValidationContext context = new ValidationContext();
    try {
        if (validator.validate(oneTimeUse, assertion, context) ==
ValidationResult.VALID) {
            return result;
        }
    } catch (Exception e) {
        return result.concat(new Saml2Error(INVALID_ASSERTION, e.getMessage()));
    }
    return result.concat(new Saml2Error(INVALID_ASSERTION,
context.getValidationFailureMessage()));
});
```

Kotlin

```
var provider = OpenSaml4AuthenticationProvider()
var validator: OneTimeUseConditionValidator = ...
provider.setAssertionValidator { assertionToken ->
    val result = OpenSaml4AuthenticationProvider
        .createDefaultAssertionValidator()
        .convert(assertionToken)
    val assertion: Assertion = assertionToken.assertion
    val oneTimeUse: OneTimeUse = assertion.conditions.oneTimeUse
    val context = ValidationContext()
    try {
        if (validator.validate(oneTimeUse, assertion, context) ==
ValidationResult.VALID) {
            return@setAssertionValidator result
        }
    } catch (e: Exception) {
        return@setAssertionValidator result.concat(Saml2Error(INVALID_ASSERTION,
e.message))
    }
    result.concat(Saml2Error(INVALID_ASSERTION, context.validationFailureMessage))
}
```

While recommended, it's not necessary to call `OpenSaml4AuthenticationProvider` 's default assertion validator. A circumstance where you would skip it would be if you don't need it to check the `<AudienceRestriction>` or the `<SubjectConfirmation>` since you are doing those yourself.

Customizing Decryption

Spring Security decrypts `<saml2:EncryptedAssertion>`, `<saml2:EncryptedAttribute>`, and `<saml2:EncryptedID>` elements automatically by using the decryption `Saml2X509Credential` instances registered in the `RelyingPartyRegistration`.

`OpenSaml4AuthenticationProvider` exposes two decryption strategies. The response decrypter is for decrypting encrypted elements of the `<saml2:Response>`, like `<saml2:EncryptedAssertion>`. The assertion decrypter is for decrypting encrypted elements of the `<saml2:Assertion>`, like `<saml2:EncryptedAttribute>` and `<saml2:EncryptedID>`.

You can replace `OpenSaml4AuthenticationProvider`'s default decryption strategy with your own. For example, if you have a separate service that decrypts the assertions in a `<saml2:Response>`, you can use it instead like so:

Java

```
MyDecryptionService decryptionService = ...;
OpenSaml4AuthenticationProvider provider = new OpenSaml4AuthenticationProvider();
provider.setResponseElementsDecrypter((responseToken) ->
decryptionService.decrypt(responseToken.getResponse()));
```

Kotlin

```
val decryptionService: MyDecryptionService = ...
val provider = OpenSaml4AuthenticationProvider()
provider.setResponseElementsDecrypter { responseToken ->
decryptionService.decrypt(responseToken.response) }
```

If you are also decrypting individual elements in a `<saml2:Assertion>`, you can customize the assertion decrypter, too:

Java

```
provider.setAssertionElementsDecrypter((assertionToken) ->
decryptionService.decrypt(assertionToken.getAssertion()));
```

Kotlin

```
provider.setAssertionElementsDecrypter { assertionToken ->
decryptionService.decrypt(assertionToken.assertion) }
```

There are two separate decrypters since assertions can be signed separately from responses. Trying to decrypt a signed assertion's elements before signature verification may invalidate the signature. If your asserting party signs the response only, then it's safe to decrypt all elements using only the response decrypter.

Using a Custom Authentication Manager

Of course, the `authenticationManager` DSL method can be also used to perform a completely custom SAML 2.0 authentication. This authentication manager should expect a `Saml2AuthenticationToken` object containing the SAML 2.0 Response XML data.

Java

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        AuthenticationManager authenticationManager = new
MySaml2AuthenticationManager(...);
        http
            .authorizeRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .saml2Login(saml2 -> saml2
                .authenticationManager(authenticationManager)
            )
        ;
    }
}
```

Kotlin

```
@EnableWebSecurity
open class SecurityConfig : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        val customAuthenticationManager: AuthenticationManager =
MySaml2AuthenticationManager(...)
        http {
            authorizeRequests {
                authorize(anyRequest, authenticated)
            }
            saml2Login {
                authenticationManager = customAuthenticationManager
            }
        }
    }
}
```

13.1.8. Using `Saml2AuthenticatedPrincipal`

With the relying party correctly configured for a given asserting party, it's ready to accept assertions. Once the relying party validates an assertion, the result is a `Saml2Authentication` with a `Saml2AuthenticatedPrincipal`.

This means that you can access the principal in your controller like so:

Java

```
@Controller
public class MainController {
    @GetMapping("/")
    public String index(@AuthenticationPrincipal Saml2AuthenticatedPrincipal
principal, Model model) {
        String email = principal.getFirstAttribute("email");
        model.setAttribute("email", email);
        return "index";
    }
}
```

Kotlin

```
@Controller
class MainController {
    @GetMapping("/")
    fun index(@AuthenticationPrincipal principal: Saml2AuthenticatedPrincipal,
model: Model): String {
        val email = principal.getFirstAttribute<String>("email")
        model.setAttribute("email", email)
        return "index"
    }
}
```

Because the SAML 2.0 specification allows for each attribute to have multiple values, you can either call `getAttribute` to get the list of attributes or `getFirstAttribute` to get the first in the list. `getFirstAttribute` is quite handy when you know that there is only one value.

13.1.9. Producing `<saml2:SPSSODescriptor>` Metadata

You can publish a metadata endpoint by adding the `Saml2MetadataFilter` to the filter chain, as you'll see below:

Java

```
Converter<HttpServletRequest, RelyingPartyRegistration>
relyingPartyRegistrationResolver =
        new
DefaultRelyingPartyRegistrationResolver(this.relyingPartyRegistrationRepository);
Saml2MetadataFilter filter = new Saml2MetadataFilter(
        relyingPartyRegistrationResolver,
        new OpenSamlMetadataResolver());

http
    // ...
    .saml2Login(withDefaults())
    .addFilterBefore(filter, Saml2WebSsoAuthenticationFilter.class);
```

Kotlin

```
val relyingPartyRegistrationResolver: Converter<HttpServletRequest,
RelyingPartyRegistration> =

DefaultRelyingPartyRegistrationResolver(this.relyingPartyRegistrationRepository)
val filter = Saml2MetadataFilter(
    relyingPartyRegistrationResolver,
    OpenSamlMetadataResolver()
)

http {
    //...
    saml2Login { }
    addFilterBefore<Saml2WebSsoAuthenticationFilter>(filter)
}
```

You can use this metadata endpoint to register your relying party with your asserting party. This is often as simple as finding the correct form field to supply the metadata endpoint.

By default, the metadata endpoint is `/saml2/service-provider-metadata/{registrationId}`. You can change this by calling the `setRequestMatcher` method on the filter:

Java

```
filter.setRequestMatcher(new
AntPathRequestMatcher("/saml2/metadata/{registrationId}", "GET"));
```

Kotlin

```
filter.setRequestMatcher(AntPathRequestMatcher("/saml2/metadata/{registrationId}",
"GET"))
```

ensuring that the `registrationId` hint is at the end of the path.

Or, if you have registered a custom relying party registration resolver in the constructor, then you can specify a path without a `registrationId` hint, like so:

Java

```
filter.setRequestMatcher(new AntPathRequestMatcher("/saml2/metadata", "GET"));
```

Kotlin

```
filter.setRequestMatcher(AntPathRequestMatcher("/saml2/metadata", "GET"))
```

13.1.10. Performing Single Logout

Spring Security does not yet support single logout.

Generally speaking, though, you can achieve this by creating and registering a custom `LogoutSuccessHandler` and `RequestMatcher`:

Java

```
http
    // ...
    .logout(logout -> logout
        .logoutSuccessHandler(myCustomSuccessHandler())
        .logoutRequestMatcher(myRequestMatcher())
    )
```

Kotlin

```
http {
    logout {
        // ...
        logoutSuccessHandler = myCustomSuccessHandler()
        logoutRequestMatcher = myRequestMatcher()
    }
}
```

The success handler will send logout requests to the asserting party.

The request matcher will detect logout requests from the asserting party.

Chapter 14. Protection Against Exploits

14.1. Cross Site Request Forgery (CSRF) for Servlet Environments

This section discusses Spring Security's Cross Site Request Forgery (CSRF) support for servlet environments.

14.1.1. Using Spring Security CSRF Protection

The steps to using Spring Security's CSRF protection are outlined below:

- Use proper HTTP verbs
- Configure CSRF Protection
- Include the CSRF Token

Use proper HTTP verbs

The first step to protecting against CSRF attacks is to ensure your website uses proper HTTP verbs. This is covered in detail in Safe Methods Must be Idempotent.

Configure CSRF Protection

The next step is to configure Spring Security's CSRF protection within your application. Spring Security's CSRF protection is enabled by default, but you may need to customize the configuration. Below are a few common customizations.

Custom CsrfTokenRepository

By default Spring Security stores the expected CSRF token in the `HttpSession` using `HttpSessionCsrfTokenRepository`. There can be cases where users will want to configure a custom `CsrfTokenRepository`. For example, it might be desirable to persist the `CsrfToken` in a cookie to support a JavaScript based application.

By default the `CookieCsrfTokenRepository` will write to a cookie named `XSRF-TOKEN` and read it from a header named `X-XSRF-TOKEN` or the HTTP parameter `_csrf`. These defaults come from AngularJS

You can configure `CookieCsrfTokenRepository` in XML using the following:

Example 147. Store CSRF Token in a Cookie with XML Configuration

```
<http>
    <!-- ... -->
    <csrf token-repository-ref="tokenRepository"/>
</http>
<b:bean id="tokenRepository"
    class="org.springframework.security.web.csrf.CookieCsrfTokenRepository"
    p:cookieHttpOnly="false"/>
```

The sample explicitly sets `cookieHttpOnly=false`. This is necessary to allow JavaScript (i.e. AngularJS) to read it. If you do not need the ability to read the cookie with JavaScript directly, it is recommended to omit `cookieHttpOnly=false` to improve security.

You can configure `CookieCsrfTokenRepository` in Java Configuration using:

Example 148. Store CSRF Token in a Cookie

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
        WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) {
        http
            .csrf(csrf -> csrf

.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
       http {
            csrf {
                csrfTokenRepository =
CookieCsrfTokenRepository.withHttpOnlyFalse()
            }
        }
    }
}
```

The sample explicitly sets `cookieHttpOnly=false`. This is necessary to allow JavaScript (i.e. AngularJS) to read it. If you do not need the ability to read the cookie with JavaScript directly, it is recommended to omit `cookieHttpOnly=false` (by using `new CookieCsrfTokenRepository()` instead) to improve security.

Disable CSRF Protection

CSRF protection is enabled by default. However, it is simple to disable CSRF protection if it makes sense for your application.

The XML configuration below will disable CSRF protection.

Example 149. Disable CSRF XML Configuration

```
<http>
    <!-- ... -->
    <csrf disabled="true"/>
</http>
```

The Java configuration below will disable CSRF protection.

Example 150. Disable CSRF

Java

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends
        WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) {
        http
            .csrf(csrf -> csrf.disable());
    }
}
```

Kotlin

```
@Configuration
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
       http {
            csrf {
                disable()
            }
        }
    }
}
```

Include the CSRF Token

In order for the synchronizer token pattern to protect against CSRF attacks, we must include the actual CSRF token in the HTTP request. This must be included in a part of the request (i.e. form parameter, HTTP header, etc) that is not automatically included in the HTTP request by the browser.

Spring Security's CsrfFilter exposes a CsrfToken as an HttpServletRequest attribute named _csrf. This means that any view technology can access the CsrfToken to expose the expected token as either a form or meta tag. Fortunately, there are integrations listed below that make including the token in form and ajax requests even easier.

Form URL Encoded

In order to post an HTML form the CSRF token must be included in the form as a hidden input. For example, the rendered HTML might look like:

Example 151. CSRF Token HTML

```
<input type="hidden"
    name="_csrf"
    value="4bfd1575-3ad1-4d21-96c7-4ef2d9f86721"/>
```

Next we will discuss various ways of including the CSRF token in a form as a hidden input.

Automatic CSRF Token Inclusion

Spring Security's CSRF support provides integration with Spring's RequestDataValueProcessor via its CsrfRequestDataValueProcessor. This means that if you leverage Spring's form tag library, Thymeleaf, or any other view technology that integrates with RequestDataValueProcessor, then forms that have an unsafe HTTP method (i.e. post) will automatically include the actual CSRF token.

csrfInput Tag

If you are using JSPs, then you can use Spring's form tag library. However, if that is not an option, you can also easily include the token with the csrfInput tag.

CsrfToken Request Attribute

If the other options for including the actual CSRF token in the request do not work, you can take advantage of the fact that the CsrfToken is exposed as an HttpServletRequest attribute named _csrf.

An example of doing this with a JSP is shown below:

Example 152. CSRF Token in Form with Request Attribute

```
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}"
    method="post">
<input type="submit"
    value="Log out" />
<input type="hidden"
    name="${_csrf.parameterName}"
    value="${_csrf.token}"/>
</form>
```

Ajax and JSON Requests

If you are using JSON, then it is not possible to submit the CSRF token within an HTTP parameter. Instead you can submit the token within a HTTP header.

In the following sections we will discuss various ways of including the CSRF token as an HTTP request header in JavaScript based applications.

Automatic Inclusion

Spring Security can easily be configured to store the expected CSRF token in a cookie. By storing the expected CSRF in a cookie, JavaScript frameworks like AngularJS will automatically include the actual CSRF token in the HTTP request headers.

Meta tags

An alternative pattern to exposing the CSRF in a cookie is to include the CSRF token within your `meta` tags. The HTML might look something like this:

Example 153. CSRF meta tag HTML

```
<html>
<head>
    <meta name="_csrf" content="4bfd1575-3ad1-4d21-96c7-4ef2d9f86721"/>
    <meta name="_csrf_header" content="X-CSRF-TOKEN"/>
    <!-- ... -->
</head>
<!-- ... -->
```

Once the meta tags contained the CSRF token, the JavaScript code would read the meta tags and include the CSRF token as a header. If you were using jQuery, this could be done with the following:

Example 154. AJAX send CSRF Token

```
$(function () {
    var token = $("meta[name='_csrf']").attr("content");
    var header = $("meta[name='_csrf_header']").attr("content");
    $(document).ajaxSend(function(e, xhr, options) {
        xhr.setRequestHeader(header, token);
    });
});
```

csrfMeta tag

If you are using JSPs a simple way to write the CSRF token to the `meta` tags is by leveraging the csrfMeta tag.

CsrfToken Request Attribute

If the other options for including the actual CSRF token in the request do not work, you can take advantage of the fact that the `CsrfToken` is exposed as an `HttpServletRequest` attribute named `_csrf`. An example of doing this with a JSP is shown below:

Example 155. CSRF meta tag JSP

```
<html>
<head>
    <meta name="_csrf" content="${_csrf.token}"/>
    <!-- default header name is X-CSRF-TOKEN -->
    <meta name="_csrf_header" content="${_csrf.headerName}"/>
    <!-- ... -->
</head>
<!-- ... -->
```

14.1.2. CSRF Considerations

There are a few special considerations to consider when implementing protection against CSRF attacks. This section discusses those considerations as it pertains to servlet environments. Refer to CSRF Considerations for a more general discussion.

Logging In

It is important to require CSRF for log in requests to protect against forging log in attempts. Spring Security's servlet support does this out of the box.

Logging Out

It is important to require CSRF for log out requests to protect against forging log out attempts. If

CSRF protection is enabled (default), Spring Security's `LogoutFilter` to only process HTTP POST. This ensures that log out requires a CSRF token and that a malicious user cannot forcibly log out your users.

The easiest approach is to use a form to log out. If you really want a link, you can use JavaScript to have the link perform a POST (i.e. maybe on a hidden form). For browsers with JavaScript that is disabled, you can optionally have the link take the user to a log out confirmation page that will perform the POST.

If you really want to use HTTP GET with logout you can do so, but remember this is generally not recommended. For example, the following Java Configuration will perform logout with the URL `/logout` is requested with any HTTP method:

Example 156. Log out with HTTP GET

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
        WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) {
        http
            .logout(logout -> logout
                .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
            );
    }
}
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
       http {
            logout {
                logoutRequestMatcher = AntPathRequestMatcher("/logout")
            }
        }
    }
}
```

CSRF and Session Timeouts

By default Spring Security stores the CSRF token in the `HttpSession`. This can lead to a situation where the session expires which means there is not an expected CSRF token to validate against.

We've already discussed general solutions to session timeouts. This section discusses the specifics of CSRF timeouts as it pertains to the servlet support.

It is simple to change storage of the expected CSRF token to be in a cookie. For details, refer to the Custom CsrfTokenRepository section.

If a token does expire, you might want to customize how it is handled by specifying a custom `AccessDeniedHandler`. The custom `AccessDeniedHandler` can process the `InvalidCsrfTokenException` any way you like. For an example of how to customize the `AccessDeniedHandler` refer to the provided links for both xml and Java configuration.

Multipart (file upload)

We have already discussed how protecting multipart requests (file uploads) from CSRF attacks causes a chicken and the egg problem. This section discusses how to implement placing the CSRF token in the body and url within a servlet application.

More information about using multipart forms with Spring can be found within the 1.1.11. Multipart Resolver section of the Spring reference and the MultipartFilter javadoc.

Place CSRF Token in the Body

We have already discussed the tradeoffs of placing the CSRF token in the body. In this section we will discuss how to configure Spring Security to read the CSRF from the body.

In order to read the CSRF token from the body, the `MultipartFilter` is specified before the Spring Security filter. Specifying the `MultipartFilter` before the Spring Security filter means that there is no authorization for invoking the `MultipartFilter` which means anyone can place temporary files on your server. However, only authorized users will be able to submit a File that is processed by your application. In general, this is the recommended approach because the temporary file upload should have a negligible impact on most servers.

To ensure `MultipartFilter` is specified before the Spring Security filter with java configuration, users can override beforeSpringSecurityFilterChain as shown below:

Example 157. Initializer MultipartFilter

Java

```
public class SecurityApplicationInitializer extends
AbstractSecurityWebApplicationInitializer {

    @Override
    protected void beforeSpringSecurityFilterChain(ServletContext servletContext)
{
        insertFilters(servletContext, new MultipartFilter());
    }
}
```

Kotlin

```
class SecurityApplicationInitializer : AbstractSecurityWebApplicationInitializer()
{
    override fun beforeSpringSecurityFilterChain(servletContext: ServletContext?)
{
        insertFilters(servletContext, MultipartFilter())
    }
}
```

To ensure `MultipartFilter` is specified before the Spring Security filter with XML configuration, users can ensure the <filter-mapping> element of the `MultipartFilter` is placed before the springSecurityFilterChain within the web.xml as shown below:

Example 158. web.xml - MultipartFilter

```
<filter>
    <filter-name>MultipartFilter</filter-name>
    <filter-
class>org.springframework.web.multipart.support.MultipartFilter</filter-class>
</filter>
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
</filter>
<filter-mapping>
    <filter-name>MultipartFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Include CSRF Token in URL

If allowing unauthorized users to upload temporary files is not acceptable, an alternative is to place the `MultipartFilter` after the Spring Security filter and include the CSRF as a query parameter in the action attribute of the form. Since the `CsrfToken` is exposed as an `HttpServletRequest` request attribute, we can use that to create an `action` with the CSRF token in it. An example with a jsp is shown below

Example 159. CSRF Token in Action

```
<form method="post"
    action="./upload?${_csrf.parameterName}=${_csrf.token}"
    enctype="multipart/form-data">
```

HiddenHttpMethodFilter

We have already discussed the trade-offs of placing the CSRF token in the body.

In Spring's Servlet support, overriding the HTTP method is done using HiddenHttpMethodFilter. More information can be found in HTTP Method Conversion section of the reference documentation.

14.2. Security HTTP Response Headers

Security HTTP Response Headers can be used to increase the security of web applications. This

section is dedicated to servlet based support for Security HTTP Response Headers.

14.2.1. Default Security Headers

Spring Security provides a default set of Security HTTP Response Headers to provide secure defaults. While each of these headers are considered best practice, it should be noted that not all clients utilize the headers, so additional testing is encouraged.

You can customize specific headers. For example, assume that you want the defaults except you wish to specify `SAMEORIGIN` for X-Frame-Options.

You can easily do this with the following Configuration:

Example 160. Customize Default Security Headers

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
        WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) {
        http
            // ...
            .headers(headers -> headers
                .frameOptions(frameOptions -> frameOptions
                    .sameOrigin()
                )
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <frame-options policy="SAMEORIGIN" />
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            // ...
            headers {
                frameOptions {
                    sameOrigin = true
                }
            }
        }
    }
}
```

If you do not want the defaults to be added and want explicit control over what should be used, you can disable the defaults. An example is provided below:

If you are using Spring Security's Configuration the following will only add Cache Control.

Example 161. Customize Cache Control Headers

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers(headers -> headers
                // do not use any default headers unless explicitly listed
                .defaultsDisabled()
                .cacheControl(withDefaults())
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers defaults-disabled="true">
        <cache-control/>
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            // ...
            headers {
                // do not use any default headers unless explicitly listed
                defaultsDisabled = true
                cacheControl {
                }
            }
        }
    }
}
```

If necessary, you can disable all of the HTTP Security response headers with the following Configuration:

Example 162. Disable All HTTP Security Headers

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers(headers -> headers.disable());
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers disabled="true" />
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            // ...
            headers {
                disable()
            }
        }
    }
}
```

14.2.2. Cache Control

Spring Security includes Cache Control headers by default.

However, if you actually want to cache specific responses, your application can selectively invoke HttpServletResponse.setHeader(String,String) to override the header set by Spring Security. This is useful to ensure things like CSS, JavaScript, and images are properly cached.

When using Spring Web MVC, this is typically done within your configuration. Details on how to do this can be found in the Static Resources portion of the Spring Reference documentation

If necessary, you can also disable Spring Security's cache control HTTP response headers.

Example 163. Cache Control Disabled

Java

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) {
        http
            // ...
            .headers(headers -> headers
                .cacheControl(cache -> cache.disable())
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <cache-control disabled="true"/>
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
       http {
            headers {
                cacheControl {
                    disable()
                }
            }
        }
    }
}
```

14.2.3. Content Type Options

Spring Security includes Content-Type headers by default. However, you can disable it with:

Example 164. Content Type Options Disabled

Java

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends
        WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) {
        http
            // ...
            .headers(headers -> headers
                .contentTypeOptions(contentTypeOptions ->
contentTypeOptions.disable())
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <content-type-options disabled="true"/>
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
       http {
            headers {
                contentTypeOptions {
                    disable()
                }
            }
        }
    }
}
```

14.2.4. HTTP Strict Transport Security (HSTS)

Spring Security provides the Strict Transport Security header by default. However, you can customize the results explicitly. For example, the following is an example of explicitly providing HSTS:

Example 165. Strict Transport Security

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers(headers -> headers
                .httpStrictTransportSecurity(hsts -> hsts
                    .includeSubDomains(true)
                    .preload(true)
                    .maxAgeInSeconds(31536000)
                )
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <hsts
            include-subdomains="true"
            max-age-seconds="31536000"
            preload="true" />
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            headers {
                httpStrictTransportSecurity {
                    includeSubDomains = true
                    preload = true
                    maxAgeInSeconds = 31536000
                }
            }
        }
    }
}
```

14.2.5. HTTP Public Key Pinning (HPKP)

For passivity reasons, Spring Security provides servlet support for HTTP Public Key Pinning but it is no longer recommended.

You can enable HPKP headers with the following Configuration:

Example 166. HTTP Public Key Pinning

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers(headers -> headers
                .httpPublicKeyPinning(hpkp -> hpkp
                    .includeSubDomains(true)
                    .reportUri("https://example.net/pkp-report")
                    .addSha256Pins("d6qzRu9zOECb90Uez27xWltNsj0e1Md7GkYYkVoZWmM=",
"E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=")
                )
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <hpkp
            include-subdomains="true"
            report-uri="https://example.net/pkp-report">
            <pins>
                <pin
algorithm="sha256">d6qzRu9zOECb90Uez27xWltNsj0e1Md7GkYYkVoZWmM=</pin>
                <pin
algorithm="sha256">E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=</pin>
            </pins>
        </hpkp>
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            headers {
                httpPublicKeyPinning {
                    includeSubDomains = true
                    reportUri = "https://example.net/pkp-report"
                    pins = mapOf("d6qzRu9zOECb90Uez27xWltNsj0e1Md7GkYYkVoZWmM=" to
"sha256",
                            "E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=" to
"sha256")
                }
            }
        }
    }
}
```

14.2.6. X-Frame-Options

By default, Spring Security disables rendering within an iframe using X-Frame-Options.

You can customize frame options to use the same origin within a Configuration using the following:

Example 167. X-Frame-Options: SAMEORIGIN

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers(headers -> headers
                .frameOptions(frameOptions -> frameOptions
                    .sameOrigin()
                )
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <frame-options
        policy="SAMEORIGIN" />
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            headers {
                frameOptions {
                    sameOrigin = true
                }
            }
        }
    }
}
```

14.2.7. X-XSS-Protection

By default, Spring Security instructs browsers to block reflected XSS attacks using the <<headers-xss-protection,X-XSS-Protection header>>. However, you can change this default. For example, the following Configuration specifies that Spring Security should no longer instruct browsers to block the content:

Example 168. X-XSS-Protection Customization

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers(headers -> headers
                .xssProtection(xss -> xss
                    .block(false)
                )
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <xss-protection block="false"/>
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        // ...
        http {
            headers {
                xssProtection {
                    block = false
                }
            }
        }
    }
}
```

14.2.8. Content Security Policy (CSP)

Spring Security does not add Content Security Policy by default, because a reasonable default is impossible to know without context of the application. The web application author must declare the security policy(s) to enforce and/or monitor for the protected resources.

For example, given the following security policy:

Example 169. Content Security Policy Example

```
Content-Security-Policy: script-src 'self' https://trustedscripts.example.com;
object-src https://trustedplugins.example.com; report-uri /csp-report-endpoint/
```

You can enable the CSP header as shown below:

Example 170. Content Security Policy

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) {
        http
            // ...
            .headers(headers -> headers
                .contentSecurityPolicy(csp -> csp
                    .policyDirectives("script-src 'self'
https://trustedscripts.example.com; object-src https://trustedplugins.example.com;
report-uri /csp-report-endpoint/")
                )
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <content-security-policy
            policy-directives="script-src 'self'
https://trustedscripts.example.com; object-src https://trustedplugins.example.com;
report-uri /csp-report-endpoint/" />
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            // ...
            headers {
                contentSecurityPolicy {
                    policyDirectives = "script-src 'self'
https://trustedscripts.example.com; object-src https://trustedplugins.example.com;
report-uri /csp-report-endpoint/"
                }
            }
        }
    }
}
```

To enable the CSP `report-only` header, provide the following configuration:

Example 171. Content Security Policy Report Only

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
        WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers(headers -> headers
                .contentSecurityPolicy(csp -> csp
                    .policyDirectives("script-src 'self'
https://trustedscripts.example.com; object-src https://trustedplugins.example.com;
report-uri /csp-report-endpoint/")
                    .reportOnly()
                )
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <content-security-policy
            policy-directives="script-src 'self'
https://trustedscripts.example.com; object-src https://trustedplugins.example.com;
report-uri /csp-report-endpoint/"
            report-only="true" />
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            // ...
            headers {
                contentSecurityPolicy {
                    policyDirectives = "script-src 'self'
https://trustedscripts.example.com; object-src https://trustedplugins.example.com;
report-uri /csp-report-endpoint/"
                    reportOnly = true
                }
            }
        }
    }
}
```

14.2.9. Referrer Policy

Spring Security does not add Referrer Policy headers by default. You can enable the Referrer Policy header using the configuration as shown below:

Example 172. Referrer Policy

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) {
        http
            // ...
            .headers(headers -> headers
                .referrerPolicy(referrer -> referrer
                    .policy(ReferrerPolicy.SAME_ORIGIN)
                )
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <referrer-policy policy="same-origin" />
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            // ...
            headers {
                referrerPolicy {
                    policy = ReferrerPolicy.SAME_ORIGIN
                }
            }
        }
    }
}
```

14.2.10. Feature Policy

Spring Security does not add Feature Policy headers by default. The following `Feature-Policy` header:

Example 173. Feature-Policy Example

```
Feature-Policy: geolocation 'self'
```

can enable the Feature Policy header using the configuration shown below:

Example 174. Feature-Policy

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers(headers -> headers
                .featurePolicy("geolocation 'self'")
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <feature-policy policy-directives="geolocation 'self'" />
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            // ...
            headers {
                featurePolicy("geolocation 'self'")
            }
        }
    }
}
```

14.2.11. Permissions Policy

Spring Security does not add Permissions Policy headers by default. The following `Permissions-Policy` header:

Example 175. Permissions-Policy Example

```
Permissions-Policy: geolocation=(self)
```

can enable the Permissions Policy header using the configuration shown below:

Example 176. Permissions-Policy

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers(headers -> headers
                .permissionsPolicy(permissions -> permissions
                    .policy("geolocation=(self)")
                )
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <permissions-policy policy="geolocation=(self)" />
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            // ...
            headers {
                permissionPolicy {
                    policy = "geolocation=(self)"
                }
            }
        }
    }
}
```

14.2.12. Clear Site Data

Spring Security does not add Clear-Site-Data headers by default. The following Clear-Site-Data header:

Example 177. Clear-Site-Data Example

```
Clear-Site-Data: "cache", "cookies"
```

can be sent on log out with the following configuration:

Example 178. Clear-Site-Data

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .logout()
                .addLogoutHandler(new HeaderWriterLogoutHandler(new
ClearSiteDataHeaderWriter(CACHE, COOKIES)));
    }
}
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            // ...
            logout {

addLogoutHandler(HeaderWriterLogoutHandler(ClearSiteDataHeaderWriter(CACHE,
COOKIES)))
            }
        }
    }
}
```

14.2.13. Custom Headers

Spring Security has mechanisms to make it convenient to add the more common security headers to your application. However, it also provides hooks to enable adding custom headers.

Static Headers

There may be times you wish to inject custom security headers into your application that are not supported out of the box. For example, given the following custom security header:

```
X-Custom-Security-Header: header-value
```

The headers could be added to the response using the following Configuration:

Example 179. StaticHeadersWriter

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers(headers -> headers
                .addHeaderWriter(new StaticHeadersWriter("X-Custom-Security-
Header","header-value"))
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <header name="X-Custom-Security-Header" value="header-value"/>
    </headers>
</http>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            // ...
            headers {
                addHeaderWriter(StaticHeadersWriter("X-Custom-Security-
Header","header-value"))
            }
        }
    }
}
```

Headers Writer

When the namespace or Java configuration does not support the headers you want, you can create

a custom `HeadersWriter` instance or even provide a custom implementation of the `HeadersWriter`.

Let's take a look at an example of using an custom instance of `XFrameOptionsHeaderWriter`. If you wanted to explicitly configure X-Frame-Options it could be done with the following Configuration:

Example 180. Headers Writer

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers(headers -> headers
                .addHeaderWriter(new
XFrameOptionsHeaderWriter(XFrameOptionsMode.SAMEORIGIN))
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <header ref="frameOptionsWriter"/>
    </headers>
</http>
<!-- Requires the c-namespace.
See https://docs.spring.io/spring/docs/current/spring-framework-
reference/htmlsingle/#beans-c-namespace
-->
<beans:bean id="frameOptionsWriter"

class="org.springframework.security.web.header.writers.frameoptions.XFrameOptionsH
eaderWriter"
    c:frameOptionsMode="SAMEORIGIN"/>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            // ...
            headers {

addHeaderWriter(XFrameOptionsHeaderWriter(XFrameOptionsMode.SAMEORIGIN))
            }
        }
    }
}
```

DelegatingRequestMatcherHeaderWriter

At times you may want to only write a header for certain requests. For example, perhaps you want to only protect your log in page from being framed. You could use the `DelegatingRequestMatcherHeaderWriter` to do so.

An example of using `DelegatingRequestMatcherHeaderWriter` in Java Configuration can be seen below:

Example 181. DelegatingRequestMatcherHeaderWriter Java Configuration

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        RequestMatcher matcher = new AntPathRequestMatcher("/login");
        DelegatingRequestMatcherHeaderWriter headerWriter =
            new DelegatingRequestMatcherHeaderWriter(matcher,new
XFrameOptionsHeaderWriter());
        http
            // ...
            .headers(headers -> headers
                .frameOptions(frameOptions -> frameOptions.disable())
                .addHeaderWriter(headerWriter)
            );
    }
}
```

XML

```
<http>
    <!-- ... -->

    <headers>
        <frame-options disabled="true"/>
        <header ref="headerWriter"/>
    </headers>
</http>

<beans:bean id="headerWriter"

class="org.springframework.security.web.header.writers.DelegatingRequestMatcherHea
derWriter">
    <beans:constructor-arg>
        <bean
class="org.springframework.security.web.util.matcher.AntPathRequestMatcher"
            c:pattern="/login"/>
    </beans:constructor-arg>
    <beans:constructor-arg>
        <beans:bean

class="org.springframework.security.web.header.writers.frameoptions.XFrameOptionsH
eaderWriter"/>
    </beans:constructor-arg>
</beans:bean>
```

Kotlin

```
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        val matcher: RequestMatcher = AntPathRequestMatcher("/login")
        val headerWriter = DelegatingRequestMatcherHeaderWriter(matcher,
XFrameOptionsHeaderWriter())
       http {
            headers {
                frameOptions {
                    disable()
                }
                addHeaderWriter(headerWriter)
            }
        }
    }
}
```

14.3. HTTP

All HTTP based communication should be protected using TLS.

Below you can find details around Servlet specific features that assist with HTTPS usage.

14.3.1. Redirect to HTTPS

If a client makes a request using HTTP rather than HTTPS, Spring Security can be configured to redirect to HTTPS.

For example, the following Java configuration will redirect any HTTP requests to HTTPS:

Example 182. Redirect to HTTPS

Java

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends
        WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) {
        http
            // ...
            .requiresChannel(channel -> channel
                .anyRequest().requiresSecure()
            );
    }
}
```

Kotlin

```
@Configuration
@EnableWebSecurity
class SecurityConfig : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        http {
            // ...
            requiresChannel {
                secure(AnyRequestMatcher.INSTANCE, "REQUIRES_SECURE_CHANNEL")
            }
        }
    }
}
```

The following XML configuration will redirect all HTTP requests to HTTPS

Example 183. Redirect to HTTPS with XML Configuration

```
<http>
    <intercept-url pattern="/**" access="ROLE_USER" requires-channel="https"/>
...
</http>
```

14.3.2. Strict Transport Security

Spring Security provides support for Strict Transport Security and enables it by default.

14.3.3. Proxy Server Configuration

Spring Security integrates with proxy servers.

14.4. HttpFirewall

Spring Security has several areas where patterns you have defined are tested against incoming requests in order to decide how the request should be handled. This occurs when the `FilterChainProxy` decides which filter chain a request should be passed through and also when the `FilterSecurityInterceptor` decides which security constraints apply to a request. It's important to understand what the mechanism is and what URL value is used when testing against the patterns that you define.

The Servlet Specification defines several properties for the `HttpServletRequest` which are accessible via getter methods, and which we might want to match against. These are the `contextPath`, `servletPath`, `pathInfo` and `queryString`. Spring Security is only interested in securing paths within the application, so the `contextPath` is ignored. Unfortunately, the servlet spec does not define exactly what the values of `servletPath` and `pathInfo` will contain for a particular request URI. For example, each path segment of a URL may contain parameters, as defined in RFC 2396 [6]. The Specification does not clearly state whether these should be included in the `servletPath` and `pathInfo` values and the behaviour varies between different servlet containers. There is a danger that when an application is deployed in a container which does not strip path parameters from these values, an attacker could add them to the requested URL in order to cause a pattern match to succeed or fail unexpectedly. [7]. Other variations in the incoming URL are also possible. For example, it could contain path-traversal sequences (like `/../`) or multiple forward slashes (`//`) which could also cause pattern-matches to fail. Some containers normalize these out before performing the servlet mapping, but others don't. To protect against issues like these, `FilterChainProxy` uses an `HttpFirewall` strategy to check and wrap the request. Un-normalized requests are automatically rejected by default, and path parameters and duplicate slashes are removed for matching purposes. [8]. It is therefore essential that a `FilterChainProxy` is used to manage the security filter chain. Note that the `servletPath` and `pathInfo` values are decoded by the container, so your application should not have any valid paths which contain semi-colons, as these parts will be removed for matching purposes.

As mentioned above, the default strategy is to use Ant-style paths for matching and this is likely to

be the best choice for most users. The strategy is implemented in the class `AntPathRequestMatcher` which uses Spring's `AntPathMatcher` to perform a case-insensitive match of the pattern against the concatenated `servletPath` and `pathInfo`, ignoring the `queryString`.

If for some reason, you need a more powerful matching strategy, you can use regular expressions. The strategy implementation is then `RegexRequestMatcher`. See the Javadoc for this class for more information.

In practice we recommend that you use method security at your service layer, to control access to your application, and do not rely entirely on the use of security constraints defined at the web-application level. URLs change and it is difficult to take account of all the possible URLs that an application might support and how requests might be manipulated. You should try and restrict yourself to using a few simple ant paths which are simple to understand. Always try to use a "deny-by-default" approach where you have a catch-all wildcard (/ **or**) defined last and denying access.

Security defined at the service layer is much more robust and harder to bypass, so you should always take advantage of Spring Security's method security options.

The `HttpFirewall` also prevents HTTP Response Splitting by rejecting new line characters in the HTTP Response headers.

By default the `StrictHttpFirewall` is used. This implementation rejects requests that appear to be malicious. If it is too strict for your needs, then you can customize what types of requests are rejected. However, it is important that you do so knowing that this can open your application up to attacks. For example, if you wish to leverage Spring MVC's Matrix Variables, the following configuration could be used:

Example 184. Allow Matrix Variables

Java

```
@Bean
public StrictHttpFirewall httpFirewall() {
    StrictHttpFirewall firewall = new StrictHttpFirewall();
    firewall.setAllowSemicolon(true);
    return firewall;
}
```

XML

```
<b:bean id="httpFirewall"
    class="org.springframework.security.web.firewall.StrictHttpFirewall"
    p:allowSemicolon="true"/>

<http-firewall ref="httpFirewall"/>
```

Kotlin

```
@Bean
fun httpFirewall(): StrictHttpFirewall {
    val firewall = StrictHttpFirewall()
    firewall.setAllowSemicolon(true)
    return firewall
}
```

The `StrictHttpFirewall` provides an allowed list of valid HTTP methods that are allowed to protect against Cross Site Tracing (XST) and HTTP Verb Tampering. The default valid methods are "DELETE", "GET", "HEAD", "OPTIONS", "PATCH", "POST", and "PUT". If your application needs to modify the valid methods, you can configure a custom `StrictHttpFirewall` bean. For example, the following will only allow HTTP "GET" and "POST" methods:

Example 185. Allow Only GET & POST

Java

```
@Bean
public StrictHttpFirewall httpFirewall() {
    StrictHttpFirewall firewall = new StrictHttpFirewall();
    firewall.setAllowedHttpMethods(Arrays.asList("GET", "POST"));
    return firewall;
}
```

XML

```
<b:bean id="httpFirewall"
      class="org.springframework.security.web.firewall.StrictHttpFirewall"
      p:allowedHttpMethods="GET,HEAD"/>

<http-firewall ref="httpFirewall"/>
```

Kotlin

```
@Bean
fun httpFirewall(): StrictHttpFirewall {
    val firewall = StrictHttpFirewall()
    firewall.setAllowedHttpMethods(listOf("GET", "POST"))
    return firewall
}
```

If you are using `new MockHttpServletRequest()` it currently creates an HTTP method as an empty String "". This is an invalid HTTP method and will be rejected by Spring Security. You can resolve this by replacing it with `new MockHttpServletRequest("GET", "")`. See SPR_16851 for an issue requesting to improve this.

If you must allow any HTTP method (not recommended), you can use `StrictHttpFirewall.setUnsafeAllowAnyHttpMethod(true)`. This will disable validation of the HTTP method entirely.

`StrictHttpFirewall` also checks header names and values and parameter names. It requires that each character have a defined code point and not be a control character.

This requirement can be relaxed or adjusted as necessary using the following methods:

- `StrictHttpFirewall#setAllowedHeaderNames(Predicate)`
- `StrictHttpFirewall#setAllowedHeaderValues(Predicate)`
- `StrictHttpFirewall#setAllowedParameterNames(Predicate)`

Also, parameter values can be controlled with `setAllowedParameterValues(Predicate)`.

For example, to switch off this check, you can wire your `StrictHttpFirewall` with `Predicate` s that always return `true`, like so:

Example 186. Allow Any Header Name, Header Value, and Parameter Name

Java

```
@Bean
public StrictHttpFirewall httpFirewall() {
    StrictHttpFirewall firewall = new StrictHttpFirewall();
    firewall.setAllowedHeaderNames((header) -> true);
    firewall.setAllowedHeaderValues((header) -> true);
    firewall.setAllowedParameterNames((parameter) -> true);
    return firewall;
}
```

Kotlin

```
@Bean
fun httpFirewall(): StrictHttpFirewall {
    val firewall = StrictHttpFirewall()
    firewall.setAllowedHeaderNames { true }
    firewall.setAllowedHeaderValues { true }
    firewall.setAllowedParameterNames { true }
    return firewall
}
```

Or, there might be a specific value that you need to allow.

For example, iPhone X uses a `User-Agent` that includes a character not in the ISO-8859-1 charset. Due to this fact, some application servers will parse this value into two separate characters, the latter being an undefined character.

You can address this with the `setAllowedHeaderValues` method, as you can see below:

Example 187. Allow Certain User Agents

Java

```
@Bean
public StrictHttpFirewall httpFirewall() {
    StrictHttpFirewall firewall = new StrictHttpFirewall();
    Pattern allowed = Pattern.compile("[\\p{IsAssigned}&&[^\\p{IsControl}]]*");
    Pattern userAgent = ...;
    firewall.setAllowedHeaderValues((header) -> allowed.matcher(header).matches()
|| userAgent.matcher(header).matches());
    return firewall;
}
```

Kotlin

```
@Bean
fun httpFirewall(): StrictHttpFirewall {
    val firewall = StrictHttpFirewall()
    val allowed = Pattern.compile("[\\p{IsAssigned}&&[^\\p{IsControl}]]*")
    val userAgent = Pattern.compile(...)
    firewall.setAllowedHeaderValues { allowed.matcher(it).matches() ||
userAgent.matcher(it).matches() }
    return firewall
}
```

In the case of header values, you may instead consider parsing them as UTF-8 at verification time like so:

Example 188. Parse Headers As UTF-8

Java

```
firewall.setAllowedHeaderValues((header) -> {
    String parsed = new String(header.getBytes(ISO_8859_1), UTF_8);
    return allowed.matcher(parsed).matches();
});
```

Kotlin

```
firewall.setAllowedHeaderValues {
    val parsed = String(header.getBytes(ISO_8859_1), UTF_8)
    return allowed.matcher(parsed).matches()
}
```

[6] You have probably seen this when a browser doesn't support cookies and the `jsessionid` parameter is appended to the URL after a semi-colon. However the RFC allows the presence of these parameters in any path segment of the URL

[7] The original values will be returned once the request leaves the `FilterChainProxy`, so will still be available to the application.

[8] So, for example, an original request path `/secure;hack=1/somefile.html;hack=2` will be returned as `/secure/somefile.html`.

Chapter 15. Integrations

15.1. Servlet API integration

This section describes how Spring Security is integrated with the Servlet API.

15.1.1. Servlet 2.5+ Integration

HttpServletRequest.getRemoteUser()

The HttpServletRequest.getRemoteUser() will return the result of `SecurityContextHolder.getContext().getAuthentication().getName()` which is typically the current username. This can be useful if you want to display the current username in your application. Additionally, checking if this is null can be used to indicate if a user has authenticated or is anonymous. Knowing if the user is authenticated or not can be useful for determining if certain UI elements should be shown or not (i.e. a log out link should only be displayed if the user is authenticated).

HttpServletRequest.getUserPrincipal()

The HttpServletRequest.getUserPrincipal() will return the result of `SecurityContextHolder.getContext().getAuthentication()`. This means it is an `Authentication` which is typically an instance of `UsernamePasswordAuthenticationToken` when using username and password based authentication. This can be useful if you need additional information about your user. For example, you might have created a custom `UserDetailsService` that returns a custom `UserDetails` containing a first and last name for your user. You could obtain this information with the following:

Java

```
Authentication auth = httpServletRequest.getUserPrincipal();
// assume integrated custom UserDetails called MyCustomUserDetails
// by default, typically instance of UserDetails
MyCustomUserDetails userDetails = (MyCustomUserDetails) auth.getPrincipal();
String firstName = userDetails.getFirstName();
String lastName = userDetails.getLastName();
```

Kotlin

```
val auth: Authentication = httpServletRequest.getUserPrincipal()
// assume integrated custom UserDetails called MyCustomUserDetails
// by default, typically instance of UserDetails
val userDetails: MyCustomUserDetails = auth.principal as MyCustomUserDetails
val firstName: String = userDetails.firstName
val lastName: String = userDetails.lastName
```

It should be noted that it is typically bad practice to perform so much logic throughout your application. Instead, one should centralize it to reduce any coupling of Spring Security and the Servlet API's.

HttpServletRequest.isUserInRole(String)

The HttpServletRequest.isUserInRole(String) will determine if `SecurityContextHolder.getContext().getAuthentication().getAuthorities()` contains a `GrantedAuthority` with the role passed into `isUserInRole(String)`. Typically users should not pass in the "ROLE_" prefix into this method since it is added automatically. For example, if you want to determine if the current user has the authority "ROLE_ADMIN", you could use the following:

Java

```
boolean isAdmin = httpServletRequest.isUserInRole("ADMIN");
```

Kotlin

```
val isAdmin: Boolean = httpServletRequest.isUserInRole("ADMIN")
```

This might be useful to determine if certain UI components should be displayed. For example, you might display admin links only if the current user is an admin.

15.1.2. Servlet 3+ Integration

The following section describes the Servlet 3 methods that Spring Security integrates with.

HttpServletRequest.authenticate(HttpServletRequest,HttpServletResponse)

The HttpServletRequest.authenticate(HttpServletRequest,HttpServletResponse) method can be used to ensure that a user is authenticated. If they are not authenticated, the configured AuthenticationEntryPoint will be used to request the user to authenticate (i.e. redirect to the login page).

HttpServletRequest.login(String,String)

The HttpServletRequest.login(String,String) method can be used to authenticate the user with the current `AuthenticationManager`. For example, the following would attempt to authenticate with the username "user" and password "password":

Java

```
try {
httpServletRequest.login("user","password");
} catch(ServletException ex) {
// fail to authenticate
}
```

Kotlin

```
try {
    httpServletRequest.login("user", "password")
} catch (ex: ServletException) {
    // fail to authenticate
}
```

It is not necessary to catch the ServletException if you want Spring Security to process the failed authentication attempt.

HttpServletRequest.logout()

The HttpServletRequest.logout() method can be used to log the current user out.

Typically this means that the SecurityContextHolder will be cleared out, the HttpSession will be invalidated, any "Remember Me" authentication will be cleaned up, etc. However, the configured LogoutHandler implementations will vary depending on your Spring Security configuration. It is important to note that after HttpServletRequest.logout() has been invoked, you are still in charge of writing a response out. Typically this would involve a redirect to the welcome page.

AsyncContext.start(Runnable)

The AsyncContext.start(Runnable) method that ensures your credentials will be propagated to the new Thread. Using Spring Security's concurrency support, Spring Security overrides the AsyncContext.start(Runnable) to ensure that the current SecurityContext is used when processing the Runnable. For example, the following would output the current user's Authentication:

Java

```
final AsyncContext async = httpServletRequest.startAsync();
async.start(new Runnable() {
    public void run() {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        try {
            final HttpServletResponse asyncResponse = (HttpServletResponse)
async.getResponse();
            asyncResponse.setStatus(HttpServletResponse.SC_OK);
            asyncResponse.getWriter().write(String.valueOf(authentication));
            async.complete();
        } catch(Exception ex) {
            throw new RuntimeException(ex);
        }
    }
});
```

Kotlin

```
val async: AsyncContext = httpServletRequest.startAsync()
async.start {
    val authentication: Authentication =
SecurityContextHolder.getContext().authentication
    try {
        val asyncResponse = async.response as HttpServletResponse
        asyncResponse.status = HttpServletResponse.SC_OK
        asyncResponse.writer.write(String.valueOf(authentication))
        async.complete()
    } catch (ex: Exception) {
        throw RuntimeException(ex)
    }
}
```

Async Servlet Support

If you are using Java Based configuration, you are ready to go. If you are using XML configuration, there are a few updates that are necessary. The first step is to ensure you have updated your web.xml to use at least the 3.0 schema as shown below:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
https://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
version="3.0">

</web-app>
```

Next you need to ensure that your springSecurityFilterChain is setup for processing asynchronous requests.

```
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
</filter-class>
<async-supported>true</async-supported>
</filter>
<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
<dispatcher>REQUEST</dispatcher>
<dispatcher>ASYNC</dispatcher>
</filter-mapping>
```

That's it! Now Spring Security will ensure that your SecurityContext is propagated on asynchronous requests too.

So how does it work? If you are not really interested, feel free to skip the remainder of this section, otherwise read on. Most of this is built into the Servlet specification, but there is a little bit of tweaking that Spring Security does to ensure things work with asynchronous requests properly. Prior to Spring Security 3.2, the SecurityContext from the SecurityContextHolder was automatically saved as soon as the HttpServletResponse was committed. This can cause issues in an Async environment. For example, consider the following:

Java

```
httpServletRequest.startAsync();
new Thread("AsyncThread") {
    @Override
    public void run() {
        try {
            // Do work
            TimeUnit.SECONDS.sleep(1);

            // Write to and commit the httpServletResponse
            httpServletResponse.getOutputStream().flush();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}.start();
```

Kotlin

```
httpServletRequest.startAsync()
object : Thread("AsyncThread") {
    override fun run() {
        try {
            // Do work
            TimeUnit.SECONDS.sleep(1)

            // Write to and commit the httpServletResponse
            httpServletResponse.outputStream.flush()
        } catch (ex: java.lang.Exception) {
            ex.printStackTrace()
        }
    }
}.start()
```

The issue is that this Thread is not known to Spring Security, so the SecurityContext is not propagated to it. This means when we commit the HttpServletResponse there is no SecurityContext. When Spring Security automatically saved the SecurityContext on committing the HttpServletResponse it would lose our logged in user.

Since version 3.2, Spring Security is smart enough to no longer automatically save the SecurityContext on committing the HttpServletResponse as soon as HttpServletRequest.startAsync() is invoked.

15.1.3. Servlet 3.1+ Integration

The following section describes the Servlet 3.1 methods that Spring Security integrates with.

HttpServletRequest#changeSessionId()

The HttpServletRequest.changeSessionId() is the default method for protecting against Session Fixation attacks in Servlet 3.1 and higher.

15.2. Spring Data Integration

Spring Security provides Spring Data integration that allows referring to the current user within your queries. It is not only useful but necessary to include the user in the queries to support paged results since filtering the results afterwards would not scale.

15.2.1. Spring Data & Spring Security Configuration

To use this support, add `org.springframework.security:spring-security-data` dependency and provide a bean of type `SecurityEvaluationContextExtension`. In Java Configuration, this would look like:

Java

```
@Bean
public SecurityEvaluationContextExtension securityEvaluationContextExtension() {
    return new SecurityEvaluationContextExtension();
}
```

Kotlin

```
@Bean
fun securityEvaluationContextExtension(): SecurityEvaluationContextExtension {
    return SecurityEvaluationContextExtension()
}
```

In XML Configuration, this would look like:

```
<bean
class="org.springframework.security.data.repository.query.SecurityEvaluationContextExt
ension"/>
```

15.2.2. Security Expressions within @Query

Now Spring Security can be used within your queries. For example:

Java

```
@Repository
public interface MessageRepository extends
PagingAndSortingRepository<Message,Long> {
    @Query("select m from Message m where m.to.id = ?#{ principal?.id }")
    Page<Message> findInbox(Pageable pageable);
}
```

Kotlin

```
@Repository
interface MessageRepository : PagingAndSortingRepository<Message?, Long?> {
    @Query("select m from Message m where m.to.id = ?#{ principal?.id }")
    fun findInbox(pageable: Pageable?): Page<Message?>?
}
```

This checks to see if the `Authentication.getPrincipal().getId()` is equal to the recipient of the `Message`. Note that this example assumes you have customized the principal to be an Object that has an id property. By exposing the `SecurityEvaluationContextExtension` bean, all of the Common Security Expressions are available within the Query.

15.3. Concurrency Support

In most environments, Security is stored on a per `Thread` basis. This means that when work is done on a new `Thread`, the `SecurityContext` is lost. Spring Security provides some infrastructure to help make this much easier for users. Spring Security provides low level abstractions for working with Spring Security in multi-threaded environments. In fact, this is what Spring Security builds on to integration with AsyncContext.start(Runnable) and Spring MVC Async Integration.

15.3.1. DelegatingSecurityContextRunnable

One of the most fundamental building blocks within Spring Security's concurrency support is the `DelegatingSecurityContextRunnable`. It wraps a delegate `Runnable` in order to initialize the `SecurityContextHolder` with a specified `SecurityContext` for the delegate. It then invokes the delegate Runnable ensuring to clear the `SecurityContextHolder` afterwards. The `DelegatingSecurityContextRunnable` looks something like this:

Java

```
public void run() {
try {
    SecurityContextHolder.setContext(securityContext);
    delegate.run();
} finally {
    SecurityContextHolder.clearContext();
}
}
```

Kotlin

```
fun run() {
    try {
        SecurityContextHolder.setContext(securityContext)
        delegate.run()
    } finally {
        SecurityContextHolder.clearContext()
    }
}
```

While very simple, it makes it seamless to transfer the SecurityContext from one Thread to another. This is important since, in most cases, the SecurityContextHolder acts on a per Thread basis. For example, you might have used Spring Security's <global-method-security> support to secure one of your services. You can now easily transfer the `SecurityContext` of the current `Thread` to the `Thread` that invokes the secured service. An example of how you might do this can be found below:

Java

```
Runnable originalRunnable = new Runnable() {
public void run() {
    // invoke secured service
}
};

SecurityContext context = SecurityContextHolder.getContext();
DelegatingSecurityContextRunnable wrappedRunnable =
    new DelegatingSecurityContextRunnable(originalRunnable, context);

new Thread(wrappedRunnable).start();
```

Kotlin

```
val originalRunnable = Runnable {
    // invoke secured service
}
val context: SecurityContext = SecurityContextHolder.getContext()
val wrappedRunnable = DelegatingSecurityContextRunnable(originalRunnable, context)

Thread(wrappedRunnable).start()
```

The code above performs the following steps:

- Creates a `Runnable` that will be invoking our secured service. Notice that it is not aware of Spring Security
- Obtains the `SecurityContext` that we wish to use from the `SecurityContextHolder` and initializes the `DelegatingSecurityContextRunnable`
- Use the `DelegatingSecurityContextRunnable` to create a Thread
- Start the Thread we created

Since it is quite common to create a `DelegatingSecurityContextRunnable` with the `SecurityContext` from the `SecurityContextHolder` there is a shortcut constructor for it. The following code is the same as the code above:

Java

```
Runnable originalRunnable = new Runnable() {
public void run() {
    // invoke secured service
}
};

DelegatingSecurityContextRunnable wrappedRunnable =
    new DelegatingSecurityContextRunnable(originalRunnable);

new Thread(wrappedRunnable).start();
```

Kotlin

```
val originalRunnable = Runnable {
    // invoke secured service
}

val wrappedRunnable = DelegatingSecurityContextRunnable(originalRunnable)

Thread(wrappedRunnable).start()
```

The code we have is simple to use, but it still requires knowledge that we are using Spring Security. In the next section we will take a look at how we can utilize `DelegatingSecurityContextExecutor` to hide the fact that we are using Spring Security.

15.3.2. DelegatingSecurityContextExecutor

In the previous section we found that it was easy to use the `DelegatingSecurityContextRunnable`, but it was not ideal since we had to be aware of Spring Security in order to use it. Let's take a look at how `DelegatingSecurityContextExecutor` can shield our code from any knowledge that we are using Spring Security.

The design of `DelegatingSecurityContextExecutor` is very similar to that of `DelegatingSecurityContextRunnable` except it accepts a delegate `Executor` instead of a delegate `Runnable`. You can see an example of how it might be used below:

Java

```
SecurityContext context = SecurityContextHolder.createEmptyContext();
Authentication authentication =
    new UsernamePasswordAuthenticationToken("user","doesnotmatter",
AuthorityUtils.createAuthorityList("ROLE_USER"));
context.setAuthentication(authentication);

SimpleAsyncTaskExecutor delegateExecutor =
    new SimpleAsyncTaskExecutor();
DelegatingSecurityContextExecutor executor =
    new DelegatingSecurityContextExecutor(delegateExecutor, context);

Runnable originalRunnable = new Runnable() {
public void run() {
    // invoke secured service
}
};

executor.execute(originalRunnable);
```

Kotlin

```
val context: SecurityContext = SecurityContextHolder.createEmptyContext()
val authentication: Authentication =
    UsernamePasswordAuthenticationToken("user", "doesnotmatter",
AuthorityUtils.createAuthorityList("ROLE_USER"))
context.authentication = authentication

val delegateExecutor = SimpleAsyncTaskExecutor()
val executor = DelegatingSecurityContextExecutor(delegateExecutor, context)

val originalRunnable = Runnable {
    // invoke secured service
}

executor.execute(originalRunnable)
```

The code performs the following steps:

- Creates the `SecurityContext` to be used for our `DelegatingSecurityContextExecutor`. Note that in this example we simply create the `SecurityContext` by hand. However, it does not matter where or how we get the `SecurityContext` (i.e. we could obtain it from the `SecurityContextHolder` if we wanted).
- Creates a delegateExecutor that is in charge of executing submitted `Runnable`s
- Finally we create a `DelegatingSecurityContextExecutor` which is in charge of wrapping any Runnable that is passed into the execute method with a `DelegatingSecurityContextRunnable`. It

then passes the wrapped Runnable to the delegateExecutor. In this instance, the same `SecurityContext` will be used for every Runnable submitted to our `DelegatingSecurityContextExecutor`. This is nice if we are running background tasks that need to be run by a user with elevated privileges.

- At this point you may be asking yourself "How does this shield my code of any knowledge of Spring Security?" Instead of creating the `SecurityContext` and the `DelegatingSecurityContextExecutor` in our own code, we can inject an already initialized instance of `DelegatingSecurityContextExecutor`.

Java

```
@Autowired
private Executor executor; // becomes an instance of our
DelegatingSecurityContextExecutor

public void submitRunnable() {
Runnable originalRunnable = new Runnable() {
    public void run() {
    // invoke secured service
    }
};
executor.execute(originalRunnable);
}
```

Kotlin

```
@Autowired
lateinit var executor: Executor // becomes an instance of our
DelegatingSecurityContextExecutor

fun submitRunnable() {
    val originalRunnable = Runnable {
        // invoke secured service
    }
    executor.execute(originalRunnable)
}
```

Now our code is unaware that the `SecurityContext` is being propagated to the `Thread`, then the `originalRunnable` is run, and then the `SecurityContextHolder` is cleared out. In this example, the same user is being used to run each thread. What if we wanted to use the user from `SecurityContextHolder` at the time we invoked `executor.execute(Runnable)` (i.e. the currently logged in user) to process `originalRunnable`? This can be done by removing the `SecurityContext` argument from our `DelegatingSecurityContextExecutor` constructor. For example:

Java

```
SimpleAsyncTaskExecutor delegateExecutor = new SimpleAsyncTaskExecutor();
DelegatingSecurityContextExecutor executor =
    new DelegatingSecurityContextExecutor(delegateExecutor);
```

Kotlin

```
val delegateExecutor = SimpleAsyncTaskExecutor()
val executor = DelegatingSecurityContextExecutor(delegateExecutor)
```

Now anytime `executor.execute(Runnable)` is executed the `SecurityContext` is first obtained by the `SecurityContextHolder` and then that `SecurityContext` is used to create our `DelegatingSecurityContextRunnable`. This means that we are running our `Runnable` with the same user that was used to invoke the `executor.execute(Runnable)` code.

15.3.3. Spring Security Concurrency Classes

Refer to the Javadoc for additional integrations with both the Java concurrent APIs and the Spring Task abstractions. They are quite self-explanatory once you understand the previous code.

- `DelegatingSecurityContextCallable`
- `DelegatingSecurityContextExecutor`
- `DelegatingSecurityContextExecutorService`
- `DelegatingSecurityContextRunnable`
- `DelegatingSecurityContextScheduledExecutorService`
- `DelegatingSecurityContextSchedulingTaskExecutor`
- `DelegatingSecurityContextAsyncTaskExecutor`
- `DelegatingSecurityContextTaskExecutor`
- `DelegatingSecurityContextTaskScheduler`

15.4. Jackson Support

Spring Security provides Jackson support for persisting Spring Security related classes. This can improve the performance of serializing Spring Security related classes when working with distributed sessions (i.e. session replication, Spring Session, etc).

To use it, register the `SecurityJackson2Modules.getModules(ClassLoader)` with `ObjectMapper` (jackson-databind):

Java

```
ObjectMapper mapper = new ObjectMapper();
ClassLoader loader = getClass().getClassLoader();
List<Module> modules = SecurityJackson2Modules.getModules(loader);
mapper.registerModules(modules);

// ... use ObjectMapper as normally ...
SecurityContext context = new SecurityContextImpl();
// ...
String json = mapper.writeValueAsString(context);
```

Kotlin

```
val mapper = ObjectMapper()
val loader = javaClass.classLoader
val modules: MutableList<Module> = SecurityJackson2Modules.getModules(loader)
mapper.registerModules(modules)

// ... use ObjectMapper as normally ...
val context: SecurityContext = SecurityContextImpl()
// ...
val json: String = mapper.writeValueAsString(context)
```

The following Spring Security modules provide Jackson support:

- spring-security-core (`CoreJackson2Module`)
- spring-security-web (`WebJackson2Module`, `WebServletJackson2Module`, `WebServerJackson2Module`)
- spring-security-oauth2-client (`OAuth2ClientJackson2Module`)
- spring-security-cas (`CasJackson2Module`)

15.5. Localization

Spring Security supports localization of exception messages that end users are likely to see. If your application is designed for English-speaking users, you don't need to do anything as by default all Security messages are in English. If you need to support other locales, everything you need to know is contained in this section.

All exception messages can be localized, including messages related to authentication failures and access being denied (authorization failures). Exceptions and logging messages that are focused on developers or system deplopers (including incorrect attributes, interface contract violations, using incorrect constructors, startup time validation, debug-level logging) are not localized and instead are hard-coded in English within Spring Security's code.

Shipping in the `spring-security-core-xx.jar` you will find an `org.springframework.security` package that in turn contains a `messages.properties` file, as well as localized versions for some common languages. This should be referred to by your `ApplicationContext`, as Spring Security classes implement Spring's `MessageSourceAware` interface and expect the message resolver to be dependency injected at application context startup time. Usually all you need to do is register a bean inside your application context to refer to the messages. An example is shown below:

```
<bean id="messageSource"
    class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
<property name="basename" value="classpath:org/springframework/security/messages"/>
</bean>
```

The `messages.properties` is named in accordance with standard resource bundles and represents the default language supported by Spring Security messages. This default file is in English.

If you wish to customize the `messages.properties` file, or support other languages, you should copy the file, rename it accordingly, and register it inside the above bean definition. There are not a large number of message keys inside this file, so localization should not be considered a major initiative. If you do perform localization of this file, please consider sharing your work with the community by logging a JIRA task and attaching your appropriately-named localized version of `messages.properties`.

Spring Security relies on Spring's localization support in order to actually lookup the appropriate message. In order for this to work, you have to make sure that the locale from the incoming request is stored in Spring's `org.springframework.context.i18n.LocaleContextHolder`. Spring MVC's `DispatcherServlet` does this for your application automatically, but since Spring Security's filters are invoked before this, the `LocaleContextHolder` needs to be set up to contain the correct `Locale` before the filters are called. You can either do this in a filter yourself (which must come before the Spring Security filters in `web.xml`) or you can use Spring's `RequestContextFilter`. Please refer to the Spring Framework documentation for further details on using localization with Spring.

The "contacts" sample application is set up to use localized messages.

15.6. Spring MVC Integration

Spring Security provides a number of optional integrations with Spring MVC. This section covers the integration in further detail.

15.6.1. @EnableWebMvcSecurity

As of Spring Security 4.0, `@EnableWebMvcSecurity` is deprecated. The replacement is `@EnableWebSecurity` which will determine adding the Spring MVC features based upon the classpath.

To enable Spring Security integration with Spring MVC add the `@EnableWebSecurity` annotation to your configuration.

Spring Security provides the configuration using Spring MVC's WebMvcConfigurer. This means that if you are using more advanced options, like integrating with `WebMvcConfigurationSupport` directly, then you will need to manually provide the Spring Security configuration.

15.6.2. MvcRequestMatcher

Spring Security provides deep integration with how Spring MVC matches on URLs with `MvcRequestMatcher`. This is helpful to ensure your Security rules match the logic used to handle your requests.

In order to use `MvcRequestMatcher` you must place the Spring Security Configuration in the same `ApplicationContext` as your `DispatcherServlet`. This is necessary because Spring Security's `MvcRequestMatcher` expects a `HandlerMappingIntrospector` bean with the name of `mvcHandlerMappingIntrospector` to be registered by your Spring MVC configuration that is used to perform the matching.

For a `web.xml` this means that you should place your configuration in the `DispatcherServlet.xml`.

```
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>

<!-- All Spring Configuration (both MVC and Security) are in /WEB-INF/spring/ -->
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/spring/*.xml</param-value>
</context-param>

<servlet>
  <servlet-name>spring</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <!-- Load from the ContextLoaderListener -->
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value></param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>spring</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

Below `WebSecurityConfiguration` in placed in the `DispatcherServlet`s `ApplicationContext`.

Java

```
public class SecurityInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected Class<?>[] getRootConfigClasses() {
    return null;
  }

  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class[] { RootConfiguration.class,
        WebMvcConfiguration.class };
  }

  @Override
  protected String[] getServletMappings() {
    return new String[] { "/" };
  }
}
```

Kotlin

```
class SecurityInitializer : AbstractAnnotationConfigDispatcherServletInitializer()
{
    override fun getRootConfigClasses(): Array<Class<*>>? {
        return null
    }

    override fun getServletConfigClasses(): Array<Class<*>> {
        return arrayOf(
            RootConfiguration::class.java,
            WebMvcConfiguration::class.java
        )
    }

    override fun getServletMappings(): Array<String> {
        return arrayOf("/")
    }
}
```

It is always recommended to provide authorization rules by matching on the `HttpServletRequest` and method security.

Providing authorization rules by matching on `HttpServletRequest` is good because it happens very early in the code path and helps reduce the attack surface. Method security ensures that if someone has bypassed the web authorization rules, that your application is still secured. This is what is known as Defence in Depth

Consider a controller that is mapped as follows:

Java

```
@RequestMapping("/admin")
public String admin() {
```

Kotlin

```
@RequestMapping("/admin")
fun admin(): String {
```

If we wanted to restrict access to this controller method to admin users, a developer can provide authorization rules by matching on the `HttpServletRequest` with the following:

Java

```
protected configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests(authorize -> authorize
            .antMatchers("/admin").hasRole("ADMIN")
        );
}
```

Kotlin

```
override fun configure(http: HttpSecurity) {
    http {
        authorizeRequests {
            authorize(AntPathRequestMatcher("/admin"), hasRole("ADMIN"))
        }
    }
}
```

or in XML

```
<http>
    <intercept-url pattern="/admin" access="hasRole('ADMIN')"/>
</http>
```

With either configuration, the URL `/admin` will require the authenticated user to be an admin user. However, depending on our Spring MVC configuration, the URL `/admin.html` will also map to our `admin()` method. Additionally, depending on our Spring MVC configuration, the URL `/admin/` will also map to our `admin()` method.

The problem is that our security rule is only protecting `/admin`. We could add additional rules for all the permutations of Spring MVC, but this would be quite verbose and tedious.

Instead, we can leverage Spring Security's `MvcRequestMatcher`. The following configuration will protect the same URLs that Spring MVC will match on by using Spring MVC to match on the URL.

Java

```
protected configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests(authorize -> authorize
            .mvcMatchers("/admin").hasRole("ADMIN")
        );
}
```

Kotlin

```
override fun configure(http: HttpSecurity) {
    http {
        authorizeRequests {
            authorize("/admin", hasRole("ADMIN"))
        }
    }
}
```

or in XML

```
<http request-matcher="mvc">
    <intercept-url pattern="/admin" access="hasRole('ADMIN')"/>
</http>
```

15.6.3. @AuthenticationPrincipal

Spring Security provides `AuthenticationPrincipalArgumentResolver` which can automatically resolve the current `Authentication.getPrincipal()` for Spring MVC arguments. By using `@EnableWebSecurity` you will automatically have this added to your Spring MVC configuration. If you use XML based

configuration, you must add this yourself. For example:

```
<mvc:annotation-driven>
        <mvc:argument-resolvers>
                <bean
class="org.springframework.security.web.method.annotation.AuthenticationPrincipalArgum
entResolver" />
        </mvc:argument-resolvers>
</mvc:annotation-driven>
```

Once `AuthenticationPrincipalArgumentResolver` is properly configured, you can be entirely decoupled from Spring Security in your Spring MVC layer.

Consider a situation where a custom `UserDetailsService` that returns an `Object` that implements `UserDetails` and your own `CustomUser Object`. The `CustomUser` of the currently authenticated user could be accessed using the following code:

Java

```
@RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser() {
    Authentication authentication =
    SecurityContextHolder.getContext().getAuthentication();
    CustomUser custom = (CustomUser) authentication == null ? null :
authentication.getPrincipal();

    // .. find messages for this user and return them ...
}
```

Kotlin

```
@RequestMapping("/messages/inbox")
open fun findMessagesForUser(): ModelAndView {
    val authentication: Authentication =
SecurityContextHolder.getContext().authentication
    val custom: CustomUser? = if (authentication as CustomUser == null) null else
authentication.principal

    // .. find messages for this user and return them ...
}
```

As of Spring Security 3.2 we can resolve the argument more directly by adding an annotation. For example:

Java

```
import org.springframework.security.core.annotation.AuthenticationPrincipal;

// ...

@RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser(@AuthenticationPrincipal CustomUser
customUser) {

    // .. find messages for this user and return them ...
}
```

Kotlin

```
@RequestMapping("/messages/inbox")
open fun findMessagesForUser(@AuthenticationPrincipal customUser: CustomUser?):
ModelAndView {

    // .. find messages for this user and return them ...
}
```

Sometimes it may be necessary to transform the principal in some way. For example, if `CustomUser` needed to be final it could not be extended. In this situation the `UserDetailsService` might returns an `Object` that implements `UserDetails` and provides a method named `getCustomUser` to access `CustomUser`. For example, it might look like:

Java

```
public class CustomUserUserDetails extends User {
        // ...
        public CustomUser getCustomUser() {
                return customUser;
        }
}
```

Kotlin

```
class CustomUserUserDetails(
    username: String?,
    password: String?,
    authorities: MutableCollection<out GrantedAuthority>?
) : User(username, password, authorities) {
    // ...
    val customUser: CustomUser? = null
}
```

We could then access the `CustomUser` using a SpEL expression that uses `Authentication.getPrincipal()` as the root object:

Java

```
import org.springframework.security.core.annotation.AuthenticationPrincipal;

// ...

@RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser(@AuthenticationPrincipal(expression =
"customUser") CustomUser customUser) {

    // .. find messages for this user and return them ...
}
```

Kotlin

```
import org.springframework.security.core.annotation.AuthenticationPrincipal

// ...

@RequestMapping("/messages/inbox")
open fun findMessagesForUser(@AuthenticationPrincipal(expression = "customUser")
customUser: CustomUser?): ModelAndView {

    // .. find messages for this user and return them ...
}
```

We can also refer to Beans in our SpEL expressions. For example, the following could be used if we were using JPA to manage our Users and we wanted to modify and save a property on the current user.

Java

```
import org.springframework.security.core.annotation.AuthenticationPrincipal;

// ...

@PutMapping("/users/self")
public ModelAndView updateName(@AuthenticationPrincipal(expression =
"@jpaEntityManager.merge(#this)") CustomUser attachedCustomUser,
        @RequestParam String firstName) {

    // change the firstName on an attached instance which will be persisted to the
database
    attachedCustomUser.setFirstName(firstName);

    // ...
}
```

Kotlin

```
import org.springframework.security.core.annotation.AuthenticationPrincipal

// ...

@PutMapping("/users/self")
open fun updateName(
    @AuthenticationPrincipal(expression = "@jpaEntityManager.merge(#this)")
attachedCustomUser: CustomUser,
    @RequestParam firstName: String?
): ModelAndView {

    // change the firstName on an attached instance which will be persisted to the
database
    attachedCustomUser.setFirstName(firstName)

    // ...
}
```

We can further remove our dependency on Spring Security by making `@AuthenticationPrincipal` a meta annotation on our own annotation. Below we demonstrate how we could do this on an annotation named `@CurrentUser`.

It is important to realize that in order to remove the dependency on Spring Security, it is the consuming application that would create `@CurrentUser`. This step is not strictly required, but assists in isolating your dependency to Spring Security to a more central location.

Java

```
@Target({ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@AuthenticationPrincipal
public @interface CurrentUser {}
```

Kotlin

```
@Target(AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@MustBeDocumented
@AuthenticationPrincipal
annotation class CurrentUser
```

Now that `@CurrentUser` has been specified, we can use it to signal to resolve our `CustomUser` of the currently authenticated user. We have also isolated our dependency on Spring Security to a single file.

Java

```
@RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser(@CurrentUser CustomUser customUser) {

    // .. find messages for this user and return them ...
}
```

Kotlin

```
@RequestMapping("/messages/inbox")
open fun findMessagesForUser(@CurrentUser customUser: CustomUser?): ModelAndView {

    // .. find messages for this user and return them ...
}
```

15.6.4. Spring MVC Async Integration

Spring Web MVC 3.2+ has excellent support for Asynchronous Request Processing. With no additional configuration, Spring Security will automatically setup the `SecurityContext` to the `Thread` that invokes a `Callable` returned by your controllers. For example, the following method will automatically have its `Callable` invoked with the `SecurityContext` that was available when the `Callable` was created:

Java

```
@RequestMapping(method=RequestMethod.POST)
public Callable<String> processUpload(final MultipartFile file) {

return new Callable<String>() {
    public Object call() throws Exception {
    // ...
    return "someView";
    }
};
}
```

Kotlin

```
@RequestMapping(method = [RequestMethod.POST])
open fun processUpload(file: MultipartFile?): Callable<String> {
    return Callable {
        // ...
        "someView"
    }
}
```

Associating SecurityContext to Callable's

More technically speaking, Spring Security integrates with `WebAsyncManager`. The `SecurityContext` that is used to process the `Callable` is the `SecurityContext` that exists on the `SecurityContextHolder` at the time `startCallableProcessing` is invoked.

There is no automatic integration with a `DeferredResult` that is returned by controllers. This is because `DeferredResult` is processed by the users and thus there is no way of automatically integrating with it. However, you can still use Concurrency Support to provide transparent integration with Spring Security.

15.6.5. Spring MVC and CSRF Integration

Automatic Token Inclusion

Spring Security will automatically include the CSRF Token within forms that use the Spring MVC form tag. For example, the following JSP:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:form="http://www.springframework.org/tags/form" version="2.0">
    <jsp:directive.page language="java" contentType="text/html" />
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <!-- ... -->

    <c:url var="logoutUrl" value="/logout"/>
    <form:form action="${logoutUrl}"
        method="post">
    <input type="submit"
        value="Log out" />
    <input type="hidden"
        name="${_csrf.parameterName}"
        value="${_csrf.token}"/>
    </form:form>

    <!-- ... -->
</html>
</jsp:root>
```

Will output HTML that is similar to the following:

```
<!-- ... -->

<form action="/context/logout" method="post">
<input type="submit" value="Log out"/>
<input type="hidden" name="_csrf" value="f81d4fae-7dec-11d0-a765-00a0c91e6bf6"/>
</form>

<!-- ... -->
```

Resolving the CsrfToken

Spring Security provides `CsrfTokenArgumentResolver` which can automatically resolve the current `CsrfToken` for Spring MVC arguments. By using `@EnableWebSecurity` you will automatically have this added to your Spring MVC configuration. If you use XML based configuration, you must add this yourself.

Once `CsrfTokenArgumentResolver` is properly configured, you can expose the `CsrfToken` to your static HTML based application.

Java

```
@RestController
public class CsrfController {

    @RequestMapping("/csrf")
    public CsrfToken csrf(CsrfToken token) {
        return token;
    }
}
```

Kotlin

```
@RestController
class CsrfController {
    @RequestMapping("/csrf")
    fun csrf(token: CsrfToken): CsrfToken {
        return token
    }
}
```

It is important to keep the `CsrfToken` a secret from other domains. This means if you are using Cross Origin Sharing (CORS), you should **NOT** expose the `CsrfToken` to any external domains.

15.7. WebSocket Security

Spring Security 4 added support for securing Spring's WebSocket support. This section describes how to use Spring Security's WebSocket support.

Direct JSR-356 Support

Spring Security does not provide direct JSR-356 support because doing so would provide little value. This is because the format is unknown, so there is little Spring can do to secure an unknown format. Additionally, JSR-356 does not provide a way to intercept messages, so security would be rather invasive.

15.7.1. WebSocket Configuration

Spring Security 4.0 has introduced authorization support for WebSockets through the Spring Messaging abstraction. To configure authorization using Java Configuration, simply extend the `AbstractSecurityWebSocketMessageBrokerConfigurer` and configure the `MessageSecurityMetadataSourceRegistry`. For example:

Java

```
@Configuration
public class WebSocketSecurityConfig
      extends AbstractSecurityWebSocketMessageBrokerConfigurer { ① ②

    protected void configureInbound(MessageSecurityMetadataSourceRegistry
messages) {
        messages
                .simpDestMatchers("/user/**").authenticated() ③
    }
}
```

Kotlin

```
@Configuration
open class WebSocketSecurityConfig :
AbstractSecurityWebSocketMessageBrokerConfigurer() { ① ②
    override fun configureInbound(messages: MessageSecurityMetadataSourceRegistry)
{
        messages.simpDestMatchers("/user/**").authenticated() ③
    }
}
```

This will ensure that:

① Any inbound CONNECT message requires a valid CSRF token to enforce Same Origin Policy

② The SecurityContextHolder is populated with the user within the simpUser header attribute for any inbound request.

③ Our messages require the proper authorization. Specifically, any inbound message that starts with "/user/" will require ROLE_USER. Additional details on authorization can be found in WebSocket Authorization

Spring Security also provides XML Namespace support for securing WebSockets. A comparable XML based configuration looks like the following:

```
<websocket-message-broker> ① ②
    ③
    <intercept-message pattern="/user/**" access="hasRole('USER')" />
</websocket-message-broker>
```

This will ensure that:

① Any inbound CONNECT message requires a valid CSRF token to enforce Same Origin Policy

② The SecurityContextHolder is populated with the user within the simpUser header attribute for

any inbound request.

③ Our messages require the proper authorization. Specifically, any inbound message that starts with "/user/" will require ROLE_USER. Additional details on authorization can be found in WebSocket Authorization

15.7.2. WebSocket Authentication

WebSockets reuse the same authentication information that is found in the HTTP request when the WebSocket connection was made. This means that the `Principal` on the `HttpServletRequest` will be handed off to WebSockets. If you are using Spring Security, the `Principal` on the `HttpServletRequest` is overridden automatically.

More concretely, to ensure a user has authenticated to your WebSocket application, all that is necessary is to ensure that you setup Spring Security to authenticate your HTTP based web application.

15.7.3. WebSocket Authorization

Spring Security 4.0 has introduced authorization support for WebSockets through the Spring Messaging abstraction. To configure authorization using Java Configuration, simply extend the `AbstractSecurityWebSocketMessageBrokerConfigurer` and configure the `MessageSecurityMetadataSourceRegistry`. For example:

Java

```
@Configuration
public class WebSocketSecurityConfig extends
AbstractSecurityWebSocketMessageBrokerConfigurer {

    @Override
    protected void configureInbound(MessageSecurityMetadataSourceRegistry
messages) {
        messages
                .nullDestMatcher().authenticated() ①
                .simpSubscribeDestMatchers("/user/queue/errors").permitAll() ②
                .simpDestMatchers("/app/**").hasRole("USER") ③
                .simpSubscribeDestMatchers("/user/**",
"/topic/friends/*").hasRole("USER") ④
                .simpTypeMatchers(MESSAGE, SUBSCRIBE).denyAll() ⑤
                .anyMessage().denyAll(); ⑥

    }
}
```

Kotlin

```
@Configuration
open class WebSocketSecurityConfig :
AbstractSecurityWebSocketMessageBrokerConfigurer() {
    override fun configureInbound(messages: MessageSecurityMetadataSourceRegistry)
{
        messages
            .nullDestMatcher().authenticated() ①
            .simpSubscribeDestMatchers("/user/queue/errors").permitAll() ②
            .simpDestMatchers("/app/**").hasRole("USER") ③
            .simpSubscribeDestMatchers("/user/**",
"/topic/friends/*").hasRole("USER") ④
            .simpTypeMatchers(MESSAGE, SUBSCRIBE).denyAll() ⑤
            .anyMessage().denyAll() ⑥
    }
}
```

This will ensure that:

① Any message without a destination (i.e. anything other than Message type of MESSAGE or SUBSCRIBE) will require the user to be authenticated

② Anyone can subscribe to /user/queue/errors

③ Any message that has a destination starting with "/app/" will be require the user to have the role ROLE_USER

④ Any message that starts with "/user/" or "/topic/friends/" that is of type SUBSCRIBE will require

ROLE_USER

⑤ Any other message of type MESSAGE or SUBSCRIBE is rejected. Due to 6 we do not need this step, but it illustrates how one can match on specific message types.

⑥ Any other Message is rejected. This is a good idea to ensure that you do not miss any messages.

Spring Security also provides XML Namespace support for securing WebSockets. A comparable XML based configuration looks like the following:

```
<websocket-message-broker>
    ①
    <intercept-message type="CONNECT" access="permitAll" />
    <intercept-message type="UNSUBSCRIBE" access="permitAll" />
    <intercept-message type="DISCONNECT" access="permitAll" />

    <intercept-message pattern="/user/queue/errors" type="SUBSCRIBE"
access="permitAll" /> ②
    <intercept-message pattern="/app/**" access="hasRole('USER')" />      ③

    ④
    <intercept-message pattern="/user/**" access="hasRole('USER')" />
    <intercept-message pattern="/topic/friends/*" access="hasRole('USER')" />

    ⑤
    <intercept-message type="MESSAGE" access="denyAll" />
    <intercept-message type="SUBSCRIBE" access="denyAll" />

    <intercept-message pattern="/**" access="denyAll" /> ⑥
</websocket-message-broker>
```

This will ensure that:

① Any message of type CONNECT, UNSUBSCRIBE, or DISCONNECT will require the user to be authenticated

② Anyone can subscribe to /user/queue/errors

③ Any message that has a destination starting with "/app/" will be require the user to have the role ROLE_USER

④ Any message that starts with "/user/" or "/topic/friends/" that is of type SUBSCRIBE will require ROLE_USER

⑤ Any other message of type MESSAGE or SUBSCRIBE is rejected. Due to 6 we do not need this step, but it illustrates how one can match on specific message types.

⑥ Any other message with a destination is rejected. This is a good idea to ensure that you do not miss any messages.

WebSocket Authorization Notes

In order to properly secure your application it is important to understand Spring's WebSocket

support.

WebSocket Authorization on Message Types

It is important to understand the distinction between SUBSCRIBE and MESSAGE types of messages and how it works within Spring.

Consider a chat application.

- The system can send notifications MESSAGE to all users through a destination of "/topic/system/notifications"
- Clients can receive notifications by SUBSCRIBE to the "/topic/system/notifications".

While we want clients to be able to SUBSCRIBE to "/topic/system/notifications", we do not want to enable them to send a MESSAGE to that destination. If we allowed sending a MESSAGE to "/topic/system/notifications", then clients could send a message directly to that endpoint and impersonate the system.

In general, it is common for applications to deny any MESSAGE sent to a destination that starts with the broker prefix (i.e. "/topic/" or "/queue/").

WebSocket Authorization on Destinations

It is also is important to understand how destinations are transformed.

Consider a chat application.

- Users can send messages to a specific user by sending a message to the destination of "/app/chat".
- The application sees the message, ensures that the "from" attribute is specified as the current user (we cannot trust the client).
- The application then sends the message to the recipient using `SimpMessageSendingOperations.convertAndSendToUser("toUser", "/queue/messages", message)`.
- The message gets turned into the destination of "/queue/user/messages-<sessionid>"

With the application above, we want to allow our client to listen to "/user/queue" which is transformed into "/queue/user/messages-<sessionid>". However, we do not want the client to be able to listen to "/queue/*" because that would allow the client to see messages for every user.

In general, it is common for applications to deny any SUBSCRIBE sent to a message that starts with the broker prefix (i.e. "/topic/" or "/queue/"). Of course we may provide exceptions to account for things like

Outbound Messages

Spring contains a section titled Flow of Messages that describes how messages flow through the system. It is important to note that Spring Security only secures the `clientInboundChannel`. Spring Security does not attempt to secure the `clientOutboundChannel`.

The most important reason for this is performance. For every message that goes in, there are

typically many more that go out. Instead of securing the outbound messages, we encourage securing the subscription to the endpoints.

15.7.4. Enforcing Same Origin Policy

It is important to emphasize that the browser does not enforce the Same Origin Policy for WebSocket connections. This is an extremely important consideration.

Why Same Origin?

Consider the following scenario. A user visits bank.com and authenticates to their account. The same user opens another tab in their browser and visits evil.com. The Same Origin Policy ensures that evil.com cannot read or write data to bank.com.

With WebSockets the Same Origin Policy does not apply. In fact, unless bank.com explicitly forbids it, evil.com can read and write data on behalf of the user. This means that anything the user can do over the webSocket (i.e. transfer money), evil.com can do on that users behalf.

Since SockJS tries to emulate WebSockets it also bypasses the Same Origin Policy. This means developers need to explicitly protect their applications from external domains when using SockJS.

Spring WebSocket Allowed Origin

Fortunately, since Spring 4.1.5 Spring's WebSocket and SockJS support restricts access to the current domain. Spring Security adds an additional layer of protection to provide defence in depth.

Adding CSRF to Stomp Headers

By default Spring Security requires the CSRF token in any CONNECT message type. This ensures that only a site that has access to the CSRF token can connect. Since only the **Same Origin** can access the CSRF token, external domains are not allowed to make a connection.

Typically we need to include the CSRF token in an HTTP header or an HTTP parameter. However, SockJS does not allow for these options. Instead, we must include the token in the Stomp headers

Applications can obtain a CSRF token by accessing the request attribute named _csrf. For example, the following will allow accessing the `CsrfToken` in a JSP:

```
var headerName = "${_csrf.headerName}";
var token = "${_csrf.token}";
```

If you are using static HTML, you can expose the `CsrfToken` on a REST endpoint. For example, the following would expose the `CsrfToken` on the URL /csrf

Java

```
@RestController
public class CsrfController {

    @RequestMapping("/csrf")
    public CsrfToken csrf(CsrfToken token) {
        return token;
    }
}
```

Kotlin

```
@RestController
class CsrfController {
    @RequestMapping("/csrf")
    fun csrf(token: CsrfToken): CsrfToken {
        return token
    }
}
```

The JavaScript can make a REST call to the endpoint and use the response to populate the headerName and the token.

We can now include the token in our Stomp client. For example:

```
...
var headers = {};
headers[headerName] = token;
stompClient.connect(headers, function(frame) {
  ...

}
```

Disable CSRF within WebSockets

If you want to allow other domains to access your site, you can disable Spring Security's protection. For example, in Java Configuration you can use the following:

Java

```
@Configuration
public class WebSocketSecurityConfig extends
AbstractSecurityWebSocketMessageBrokerConfigurer {

    ...

    @Override
    protected boolean sameOriginDisabled() {
        return true;
    }
}
```

Kotlin

```
@Configuration
open class WebSocketSecurityConfig :
AbstractSecurityWebSocketMessageBrokerConfigurer() {

    // ...

    override fun sameOriginDisabled(): Boolean {
        return true
    }
}
```

15.7.5. Working with SockJS

SockJS provides fallback transports to support older browsers. When using the fallback options we need to relax a few security constraints to allow SockJS to work with Spring Security.

SockJS & frame-options

SockJS may use an transport that leverages an iframe. By default Spring Security will deny the site from being framed to prevent Clickjacking attacks. To allow SockJS frame based transports to work, we need to configure Spring Security to allow the same origin to frame the content.

You can customize X-Frame-Options with the frame-options element. For example, the following will instruct Spring Security to use "X-Frame-Options: SAMEORIGIN" which allows iframes within the same domain:

```
<http>
    <!-- ... -->

    <headers>
        <frame-options
          policy="SAMEORIGIN" />
    </headers>
</http>
```

Similarly, you can customize frame options to use the same origin within Java Configuration using the following:

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends
   WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers(headers -> headers
                .frameOptions(frameOptions -> frameOptions
                     .sameOrigin()
                )
        );
    }
}
```

Kotlin

```
@EnableWebSecurity
open class WebSecurityConfig : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            // ...
            headers {
                frameOptions {
                    sameOrigin = true
                }
            }
        }
    }
}
```

SockJS & Relaxing CSRF

SockJS uses a POST on the CONNECT messages for any HTTP based transport. Typically we need to include the CSRF token in an HTTP header or an HTTP parameter. However, SockJS does not allow for these options. Instead, we must include the token in the Stomp headers as described in Adding CSRF to Stomp Headers.

It also means we need to relax our CSRF protection with the web layer. Specifically, we want to disable CSRF protection for our connect URLs. We do NOT want to disable CSRF protection for every URL. Otherwise our site will be vulnerable to CSRF attacks.

We can easily achieve this by providing a CSRF RequestMatcher. Our Java Configuration makes this extremely easy. For example, if our stomp endpoint is "/chat" we can disable CSRF protection for only URLs that start with "/chat/" using the following configuration:

Java

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf(csrf -> csrf
                // ignore our stomp endpoints since they are protected using Stomp
headers
                .ignoringAntMatchers("/chat/**")
            )
            .headers(headers -> headers
                // allow same origin to frame our site to support iframe SockJS
                .frameOptions(frameOptions -> frameOptions
                    .sameOrigin()
                )
            )
            .authorizeRequests(authorize -> authorize
                ...
            )
            ...
```

Kotlin

```
@Configuration
@EnableWebSecurity
open class WebSecurityConfig : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            csrf {
                ignoringAntMatchers("/chat/**")
            }
            headers {
                frameOptions {
                    sameOrigin = true
                }
            }
            authorizeRequests {
                // ...
            }
            // ...
```

If we are using XML based configuration, we can use the csrf@request-matcher-ref. For example:

```
<http ...>
    <csrf request-matcher-ref="csrfMatcher"/>

    <headers>
        <frame-options policy="SAMEORIGIN"/>
    </headers>

    ...
</http>

<b:bean id="csrfMatcher"
    class="AndRequestMatcher">
    <b:constructor-arg
value="#{T(org.springframework.security.web.csrf.CsrfFilter).DEFAULT_CSRF_MATCHER}"/>
    <b:constructor-arg>
        <b:bean
class="org.springframework.security.web.util.matcher.NegatedRequestMatcher">
          <b:bean
class="org.springframework.security.web.util.matcher.AntPathRequestMatcher">
            <b:constructor-arg value="/chat/**"/>
          </b:bean>
        </b:bean>
    </b:constructor-arg>
</b:bean>
```

15.8. CORS

Spring Framework provides first class support for CORS. CORS must be processed before Spring Security because the pre-flight request will not contain any cookies (i.e. the `JSESSIONID`). If the request does not contain any cookies and Spring Security is first, the request will determine the user is not authenticated (since there are no cookies in the request) and reject it.

The easiest way to ensure that CORS is handled first is to use the `CorsFilter`. Users can integrate the `CorsFilter` with Spring Security by providing a `CorsConfigurationSource` using the following:

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // by default uses a Bean by the name of corsConfigurationSource
            .cors(withDefaults())
            ...
    }

    @Bean
    CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Arrays.asList("https://example.com"));
        configuration.setAllowedMethods(Arrays.asList("GET","POST"));
        UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }
}
```

Kotlin

```
@EnableWebSecurity
open class WebSecurityConfig : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            // by default uses a Bean by the name of corsConfigurationSource
            cors { }
            // ...
        }
    }

    @Bean
    open fun corsConfigurationSource(): CorsConfigurationSource {
        val configuration = CorsConfiguration()
        configuration.allowedOrigins = listOf("https://example.com")
        configuration.allowedMethods = listOf("GET", "POST")
        val source = UrlBasedCorsConfigurationSource()
        source.registerCorsConfiguration("/**", configuration)
        return source
    }
}
```

or in XML

```
<http>
    <cors configuration-source-ref="corsSource"/>
    ...
</http>
<b:bean id="corsSource"
class="org.springframework.web.cors.UrlBasedCorsConfigurationSource">
    ...
</b:bean>
```

If you are using Spring MVC's CORS support, you can omit specifying the `CorsConfigurationSource` and Spring Security will leverage the CORS configuration provided to Spring MVC.

Java

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // if Spring MVC is on classpath and no CorsConfigurationSource is
provided,
            // Spring Security will use CORS configuration provided to Spring MVC
            .cors(withDefaults())
            ...
    }
}
```

Kotlin

```
@EnableWebSecurity
open class WebSecurityConfig : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http {
            // if Spring MVC is on classpath and no CorsConfigurationSource is
provided,
            // Spring Security will use CORS configuration provided to Spring MVC
            cors { }
            // ...
        }
    }
}
```

or in XML

```
<http>
    <!-- Default to Spring MVC's CORS configuration -->
    <cors />
    ...
</http>
```

15.9. JSP Tag Libraries

Spring Security has its own taglib which provides basic support for accessing security information and applying security constraints in JSPs.

15.9.1. Declaring the Taglib

To use any of the tags, you must have the security taglib declared in your JSP:

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
```

15.9.2. The authorize Tag

This tag is used to determine whether its contents should be evaluated or not. In Spring Security 3.0, it can be used in two ways [9]. The first approach uses a web-security expression, specified in the `access` attribute of the tag. The expression evaluation will be delegated to the `SecurityExpressionHandler<FilterInvocation>` defined in the application context (you should have web expressions enabled in your `<http>` namespace configuration to make sure this service is available). So, for example, you might have

```
<sec:authorize access="hasRole('supervisor')">

This content will only be visible to users who have the "supervisor" authority in
their list of <tt>GrantedAuthority</tt>s.

</sec:authorize>
```

When used in conjunction with Spring Security's PermissionEvaluator, the tag can also be used to check permissions. For example:

```
<sec:authorize access="hasPermission(#domain,'read') or
hasPermission(#domain,'write')">

This content will only be visible to users who have read or write permission to the
Object found as a request attribute named "domain".

</sec:authorize>
```

A common requirement is to only show a particular link, if the user is actually allowed to click it. How can we determine in advance whether something will be allowed? This tag can also operate in an alternative mode which allows you to define a particular URL as an attribute. If the user is allowed to invoke that URL, then the tag body will be evaluated, otherwise it will be skipped. So you might have something like

```
<sec:authorize url="/admin">

This content will only be visible to users who are authorized to send requests to the
"/admin" URL.

</sec:authorize>
```

To use this tag there must also be an instance of `WebInvocationPrivilegeEvaluator` in your application context. If you are using the namespace, one will automatically be registered. This is an instance of `DefaultWebInvocationPrivilegeEvaluator`, which creates a dummy web request for the supplied URL and invokes the security interceptor to see whether the request would succeed or fail. This allows you to delegate to the access-control setup you defined using `intercept-url` declarations within the `<http>` namespace configuration and saves having to duplicate the information (such as the required roles) within your JSPs. This approach can also be combined with a `method` attribute, supplying the HTTP method, for a more specific match.

The Boolean result of evaluating the tag (whether it grants or denies access) can be stored in a page context scope variable by setting the `var` attribute to the variable name, avoiding the need for duplicating and re-evaluating the condition at other points in the page.

Disabling Tag Authorization for Testing

Hiding a link in a page for unauthorized users doesn't prevent them from accessing the URL. They could just type it into their browser directly, for example. As part of your testing process, you may want to reveal the hidden areas in order to check that links really are secured at the back end. If you set the system property `spring.security.disableUISecurity` to `true`, the `authorize` tag will still run but will not hide its contents. By default it will also surround the content with `<span class="securityHiddenUI">···</span>` tags. This allows you to display "hidden" content with a particular CSS style such as a different background colour. Try running the "tutorial" sample application with this property enabled, for example.

You can also set the properties `spring.security.securedUIPrefix` and `spring.security.securedUISuffix` if you want to change surrounding text from the default `span` tags (or use empty strings to remove it completely).

15.9.3. The authentication Tag

This tag allows access to the current `Authentication` object stored in the security context. It renders a property of the object directly in the JSP. So, for example, if the `principal` property of the `Authentication` is an instance of Spring Security's `UserDetails` object, then using `<sec:authentication property="principal.username" />` will render the name of the current user.

Of course, it isn't necessary to use JSP tags for this kind of thing and some people prefer to keep as little logic as possible in the view. You can access the `Authentication` object in your MVC controller (by calling `SecurityContextHolder.getContext().getAuthentication()`) and add the data directly to your model for rendering by the view.

15.9.4. The accesscontrollist Tag

This tag is only valid when used with Spring Security's ACL module. It checks a comma-separated list of required permissions for a specified domain object. If the current user has all of those permissions, then the tag body will be evaluated. If they don't, it will be skipped. An example might be

In general this tag should be considered deprecated. Instead use the The authorize Tag.

```
<sec:accesscontrollist hasPermission="1,2" domainObject="${someObject}">

This will be shown if the user has all of the permissions represented by the values
"1" or "2" on the given object.

</sec:accesscontrollist>
```

The permissions are passed to the `PermissionFactory` defined in the application context, converting them to ACL `Permission` instances, so they may be any format which is supported by the factory - they don't have to be integers, they could be strings like `READ` or `WRITE`. If no `PermissionFactory` is found, an instance of `DefaultPermissionFactory` will be used. The `AclService` from the application context will be used to load the `Acl` instance for the supplied object. The `Acl` will be invoked with the required permissions to check if all of them are granted.

This tag also supports the `var` attribute, in the same way as the `authorize` tag.

15.9.5. The csrfInput Tag

If CSRF protection is enabled, this tag inserts a hidden form field with the correct name and value for the CSRF protection token. If CSRF protection is not enabled, this tag outputs nothing.

Normally Spring Security automatically inserts a CSRF form field for any `<form:form>` tags you use, but if for some reason you cannot use `<form:form>`, `csrfInput` is a handy replacement.

You should place this tag within an HTML `<form></form>` block, where you would normally place other input fields. Do NOT place this tag within a Spring `<form:form></form:form>` block. Spring Security handles Spring forms automatically.

```
<form method="post" action="/do/something">
    <sec:csrfInput />
    Name:<br />
    <input type="text" name="name" />
    ...
</form>
```

15.9.6. The csrfMetaTags Tag

If CSRF protection is enabled, this tag inserts meta tags containing the CSRF protection token form field and header names and CSRF protection token value. These meta tags are useful for employing CSRF protection within JavaScript in your applications.

You should place `csrfMetaTags` within an HTML `<head></head>` block, where you would normally place other meta tags. Once you use this tag, you can access the form field name, header name, and token value easily using JavaScript. JQuery is used in this example to make the task easier.

```
<!DOCTYPE html>
<html>
    <head>
        <title>CSRF Protected JavaScript Page</title>
        <meta name="description" content="This is the description for this page" />
        <sec:csrfMetaTags />
        <script type="text/javascript" language="javascript">

            var csrfParameter = $("meta[name='_csrf_parameter']").attr("content");
            var csrfHeader = $("meta[name='_csrf_header']").attr("content");
            var csrfToken = $("meta[name='_csrf']").attr("content");

            // using XMLHttpRequest directly to send an x-www-form-urlencoded request
            var ajax = new XMLHttpRequest();
            ajax.open("POST", "https://www.example.org/do/something", true);
            ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded
data");
            ajax.send(csrfParameter + "=" + csrfToken + "&name=John&...");

            // using XMLHttpRequest directly to send a non-x-www-form-urlencoded
request
            var ajax = new XMLHttpRequest();
            ajax.open("POST", "https://www.example.org/do/something", true);
            ajax.setRequestHeader(csrfHeader, csrfToken);
            ajax.send("...");

            // using JQuery to send an x-www-form-urlencoded request
            var data = {};
            data[csrfParameter] = csrfToken;
            data["name"] = "John";
            ...
            $.ajax({
```

```
                url: "https://www.example.org/do/something",
                type: "POST",
                data: data,
                ...
            });

            // using JQuery to send a non-x-www-form-urlencoded request
            var headers = {};
            headers[csrfHeader] = csrfToken;
            $.ajax({
                url: "https://www.example.org/do/something",
                type: "POST",
                headers: headers,
                ...
            });

        <script>
    </head>
    <body>
        ...
    </body>
</html>
```

If CSRF protection is not enabled, `csrfMetaTags` outputs nothing.

[9] The legacy options from Spring Security 2.0 are also supported, but discouraged.

Chapter 16. Java Configuration

General support for Java Configuration was added to Spring Framework in Spring 3.1. Since Spring Security 3.2 there has been Spring Security Java Configuration support which enables users to easily configure Spring Security without the use of any XML.

If you are familiar with the Security Namespace Configuration then you should find quite a few similarities between it and the Security Java Configuration support.

Spring Security provides lots of sample applications which demonstrate the use of Spring Security Java Configuration.

16.1. Hello Web Security Java Configuration

The first step is to create our Spring Security Java Configuration. The configuration creates a Servlet Filter known as the `springSecurityFilterChain` which is responsible for all the security (protecting the application URLs, validating submitted username and passwords, redirecting to the log in form, etc) within your application. You can find the most basic example of a Spring Security Java Configuration below:

```
import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.*;
import org.springframework.security.config.annotation.authentication.builders.*;
import org.springframework.security.config.annotation.web.configuration.*;

@EnableWebSecurity
public class WebSecurityConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();

manager.createUser(User.withDefaultPasswordEncoder().username("user").password("passwo
rd").roles("USER").build());
        return manager;
    }
}
```

There really isn't much to this configuration, but it does a lot. You can find a summary of the features below:

- Require authentication to every URL in your application
- Generate a login form for you
- Allow the user with the **Username** *user* and the **Password** *password* to authenticate with form based authentication

- Allow the user to logout
- CSRF attack prevention
- Session Fixation protection
- Security Header integration
  - HTTP Strict Transport Security for secure requests
  - X-Content-Type-Options integration
  - Cache Control (can be overridden later by your application to allow caching of your static resources)
  - X-XSS-Protection integration
  - X-Frame-Options integration to help prevent Clickjacking
- Integrate with the following Servlet API methods
  - HttpServletRequest#getRemoteUser()
  - HttpServletRequest#getUserPrincipal()
  - HttpServletRequest#isUserInRole(java.lang.String)
  - HttpServletRequest#login(java.lang.String, java.lang.String)
  - HttpServletRequest#logout()

16.1.1. AbstractSecurityWebApplicationInitializer

The next step is to register the `springSecurityFilterChain` with the war. This can be done in Java Configuration with Spring's WebApplicationInitializer support in a Servlet 3.0+ environment. Not suprisingly, Spring Security provides a base class `AbstractSecurityWebApplicationInitializer` that will ensure the `springSecurityFilterChain` gets registered for you. The way in which we use `AbstractSecurityWebApplicationInitializer` differs depending on if we are already using Spring or if Spring Security is the only Spring component in our application.

- AbstractSecurityWebApplicationInitializer without Existing Spring - Use these instructions if you are not using Spring already
- AbstractSecurityWebApplicationInitializer with Spring MVC - Use these instructions if you are already using Spring

16.1.2. AbstractSecurityWebApplicationInitializer without Existing Spring

If you are not using Spring or Spring MVC, you will need to pass in the `WebSecurityConfig` into the superclass to ensure the configuration is picked up. You can find an example below:

```
import org.springframework.security.web.context.*;

public class SecurityWebApplicationInitializer
    extends AbstractSecurityWebApplicationInitializer {

    public SecurityWebApplicationInitializer() {
        super(WebSecurityConfig.class);
    }
}
```

The `SecurityWebApplicationInitializer` will do the following things:

- Automatically register the springSecurityFilterChain Filter for every URL in your application
- Add a ContextLoaderListener that loads the WebSecurityConfig.

16.1.3. AbstractSecurityWebApplicationInitializer with Spring MVC

If we were using Spring elsewhere in our application we probably already had a `WebApplicationInitializer` that is loading our Spring Configuration. If we use the previous configuration we would get an error. Instead, we should register Spring Security with the existing `ApplicationContext`. For example, if we were using Spring MVC our `SecurityWebApplicationInitializer` would look something like the following:

```
import org.springframework.security.web.context.*;

public class SecurityWebApplicationInitializer
    extends AbstractSecurityWebApplicationInitializer {

}
```

This would simply only register the springSecurityFilterChain Filter for every URL in your application. After that we would ensure that `WebSecurityConfig` was loaded in our existing ApplicationInitializer. For example, if we were using Spring MVC it would be added in the `getRootConfigClasses()`

```
public class MvcWebApplicationInitializer extends
        AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { WebSecurityConfig.class };
    }

    // ... other overrides ...
}
```

16.2. HttpSecurity

Thus far our WebSecurityConfig only contains information about how to authenticate our users. How does Spring Security know that we want to require all users to be authenticated? How does Spring Security know we want to support form based authentication? Actually, there is a configuration class that is being invoked behind the scenes called `WebSecurityConfigurerAdapter`. It has a method called `configure` with the following default implementation:

```
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests(authorize -> authorize
            .anyRequest().authenticated()
        )
        .formLogin(withDefaults())
        .httpBasic(withDefaults());
}
```

The default configuration above:

- Ensures that any request to our application requires the user to be authenticated
- Allows users to authenticate with form based login
- Allows users to authenticate with HTTP Basic authentication

You will notice that this configuration is quite similar the XML Namespace configuration:

```
<http>
    <intercept-url pattern="/**" access="authenticated"/>
    <form-login />
    <http-basic />
</http>
```

16.3. Multiple HttpSecurity

We can configure multiple HttpSecurity instances just as we can have multiple `<http>` blocks. The key is to extend the `WebSecurityConfigurerAdapter` multiple times. For example, the following is an example of having a different configuration for URL's that start with `/api/`.

```
@EnableWebSecurity
public class MultiHttpSecurityConfig {
    @Bean                                                                    ①
    public UserDetailsService userDetailsService() throws Exception {
        // ensure the passwords are encoded properly
        UserBuilder users = User.withDefaultPasswordEncoder();
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();

manager.createUser(users.username("user").password("password").roles("USER").build());

manager.createUser(users.username("admin").password("password").roles("USER","ADMIN").
build());
        return manager;
    }

    @Configuration
    @Order(1)                                                                ②
    public static class ApiWebSecurityConfigurationAdapter extends
WebSecurityConfigurerAdapter {
        protected void configure(HttpSecurity http) throws Exception {
            http
                .antMatcher("/api/**")                                       ③
                .authorizeRequests(authorize -> authorize
                    .anyRequest().hasRole("ADMIN")
                )
                .httpBasic(withDefaults());
        }
    }

    @Configuration                                                           ④
    public static class FormLoginWebSecurityConfigurerAdapter extends
WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests(authorize -> authorize
                    .anyRequest().authenticated()
                )
                .formLogin(withDefaults());
        }
    }
}
```

① Configure Authentication as normal

② Create an instance of `WebSecurityConfigurerAdapter` that contains `@Order` to specify which `WebSecurityConfigurerAdapter` should be considered first.

③ The `http.antMatcher` states that this `HttpSecurity` will only be applicable to URLs that start with `/api/`

④ Create another instance of `WebSecurityConfigurerAdapter`. If the URL does not start with `/api/` this configuration will be used. This configuration is considered after `ApiWebSecurityConfigurationAdapter` since it has an `@Order` value after `1` (no `@Order` defaults to last).

16.4. Custom DSLs

You can provide your own custom DSLs in Spring Security. For example, you might have something that looks like this:

```
public class MyCustomDsl extends AbstractHttpConfigurer<MyCustomDsl, HttpSecurity> {
    private boolean flag;

    @Override
    public void init(HttpSecurity http) throws Exception {
        // any method that adds another configurer
        // must be done in the init method
        http.csrf().disable();
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {
        ApplicationContext context = http.getSharedObject(ApplicationContext.class);

        // here we lookup from the ApplicationContext. You can also just create a new
instance.
        MyFilter myFilter = context.getBean(MyFilter.class);
        myFilter.setFlag(flag);
        http.addFilterBefore(myFilter, UsernamePasswordAuthenticationFilter.class);
    }

    public MyCustomDsl flag(boolean value) {
        this.flag = value;
        return this;
    }

    public static MyCustomDsl customDsl() {
        return new MyCustomDsl();
    }
}
```

This is actually how methods like `HttpSecurity.authorizeRequests()` are implemented.

The custom DSL can then be used like this:

```
@EnableWebSecurity
public class Config extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .apply(customDsl())
                .flag(true)
                .and()
            ...;
    }
}
```

The code is invoked in the following order:

- Code in `Config`s configure method is invoked
- Code in `MyCustomDsl`s init method is invoked
- Code in `MyCustomDsl`s configure method is invoked

If you want, you can have `WebSecurityConfigurerAdapter` add `MyCustomDsl` by default by using `SpringFactories`. For example, you would create a resource on the classpath named `META-INF/spring.factories` with the following contents:

META-INF/spring.factories

```
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer
= sample.MyCustomDsl
```

Users wishing to disable the default can do so explicitly.

```
@EnableWebSecurity
public class Config extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .apply(customDsl()).disable()
            ...;
    }
}
```

16.5. Post Processing Configured Objects

Spring Security's Java Configuration does not expose every property of every object that it configures. This simplifies the configuration for a majority of users. Afterall, if every property was exposed, users could use standard bean configuration.

While there are good reasons to not directly expose every property, users may still need more

advanced configuration options. To address this Spring Security introduces the concept of an `ObjectPostProcessor` which can be used to modify or replace many of the Object instances created by the Java Configuration. For example, if you wanted to configure the `filterSecurityPublishAuthorizationSuccess` property on `FilterSecurityInterceptor` you could use the following:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests(authorize -> authorize
            .anyRequest().authenticated()
            .withObjectPostProcessor(new
ObjectPostProcessor<FilterSecurityInterceptor>() {
                public <O extends FilterSecurityInterceptor> O postProcess(
                        O fsi) {
                    fsi.setPublishAuthorizationSuccess(true);
                    return fsi;
                }
            })
        );
}
```

Chapter 17. Kotlin Configuration

Spring Security Kotlin Configuration support has been available since Spring Security 5.3. It enables users to easily configure Spring Security using a native Kotlin DSL.

Spring Security provides a sample application which demonstrates the use of Spring Security Kotlin Configuration.

17.1. HttpSecurity

How does Spring Security know that we want to require all users to be authenticated? How does Spring Security know we want to support form based authentication? There is a configuration class that is being invoked behind the scenes called `WebSecurityConfigurerAdapter`. It has a method called `configure` with the following default implementation:

```
fun configure(http: HttpSecurity) {
   http {
        authorizeRequests {
            authorize(anyRequest, authenticated)
        }
       formLogin { }
       httpBasic { }
    }
}
```

The default configuration above:

- Ensures that any request to our application requires the user to be authenticated
- Allows users to authenticate with form based login
- Allows users to authenticate with HTTP Basic authentication

You will notice that this configuration is quite similar the XML Namespace configuration:

```
<http>
    <intercept-url pattern="/**" access="authenticated"/>
    <form-login />
    <http-basic />
</http>
```

17.2. Multiple HttpSecurity

We can configure multiple HttpSecurity instances just as we can have multiple `<http>` blocks. The key is to extend the `WebSecurityConfigurerAdapter` multiple times. For example, the following is an example of having a different configuration for URL's that start with `/api/`.

```
@EnableWebSecurity
class MultiHttpSecurityConfig {
    @Bean                                                                    ①
    public fun userDetailsService(): UserDetailsService {
        val users: User.UserBuilder = User.withDefaultPasswordEncoder()
        val manager = InMemoryUserDetailsManager()

manager.createUser(users.username("user").password("password").roles("USER").build())

manager.createUser(users.username("admin").password("password").roles("USER","ADMIN").
build())
        return manager
    }

    @Configuration
    @Order(1)                                                                ②
    class ApiWebSecurityConfigurationAdapter: WebSecurityConfigurerAdapter() {
        override fun configure(http: HttpSecurity) {
            http {
                securityMatcher("/api/**")                                   ③
                authorizeRequests {
                    authorize(anyRequest, hasRole("ADMIN"))
                }
                httpBasic { }
            }
        }
    }

    @Configuration                                                           ④
    class FormLoginWebSecurityConfigurerAdapter: WebSecurityConfigurerAdapter() {
        override fun configure(http: HttpSecurity) {
            http {
                authorizeRequests {
                    authorize(anyRequest, authenticated)
                }
                formLogin { }
            }
        }
    }
}
```

① Configure Authentication as normal

② Create an instance of `WebSecurityConfigurerAdapter` that contains `@Order` to specify which `WebSecurityConfigurerAdapter` should be considered first.

③ The `http.antMatcher` states that this `HttpSecurity` will only be applicable to URLs that start with `/api/`

④ Create another instance of `WebSecurityConfigurerAdapter`. If the URL does not start with `/api/` this configuration will be used. This configuration is considered after `ApiWebSecurityConfigurationAdapter` since it has an `@Order` value after `1` (no `@Order` defaults to

last).

Chapter 18. Security Namespace Configuration

18.1. Introduction

Namespace configuration has been available since version 2.0 of the Spring Framework. It allows you to supplement the traditional Spring beans application context syntax with elements from additional XML schema. You can find more information in the Spring Reference Documentation. A namespace element can be used simply to allow a more concise way of configuring an individual bean or, more powerfully, to define an alternative configuration syntax which more closely matches the problem domain and hides the underlying complexity from the user. A simple element may conceal the fact that multiple beans and processing steps are being added to the application context. For example, adding the following element from the security namespace to an application context will start up an embedded LDAP server for testing use within the application:

```
<security:ldap-server />
```

This is much simpler than wiring up the equivalent Apache Directory Server beans. The most common alternative configuration requirements are supported by attributes on the `ldap-server` element and the user is isolated from worrying about which beans they need to create and what the bean property names are. [10]. Use of a good XML editor while editing the application context file should provide information on the attributes and elements that are available. We would recommend that you try out the Eclipse IDE with Spring Tools as it has special features for working with standard Spring namespaces.

To start using the security namespace in your application context, you need to have the `spring-security-config` jar on your classpath. Then all you need to do is add the schema declaration to your application context file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:security="http://www.springframework.org/schema/security"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/security
        https://www.springframework.org/schema/security/spring-security.xsd">
    ...
</beans>
```

In many of the examples you will see (and in the sample applications), we will often use "security" as the default namespace rather than "beans", which means we can omit the prefix on all the security namespace elements, making the content easier to read. You may also want to do this if you have your application context divided up into separate files and have most of your security configuration in one of them. Your security application context file would then start like this

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/security
        https://www.springframework.org/schema/security/spring-security.xsd">
    ...
</beans:beans>
```

We'll assume this syntax is being used from now on in this chapter.

18.1.1. Design of the Namespace

The namespace is designed to capture the most common uses of the framework and provide a simplified and concise syntax for enabling them within an application. The design is based around the large-scale dependencies within the framework, and can be divided up into the following areas:

- *Web/HTTP Security* - the most complex part. Sets up the filters and related service beans used to apply the framework authentication mechanisms, to secure URLs, render login and error pages and much more.
- *Business Object (Method) Security* - options for securing the service layer.
- *AuthenticationManager* - handles authentication requests from other parts of the framework.
- *AccessDecisionManager* - provides access decisions for web and method security. A default one will be registered, but you can also choose to use a custom one, declared using normal Spring bean syntax.
- *AuthenticationProvider*s - mechanisms against which the authentication manager authenticates users. The namespace provides supports for several standard options and also a means of adding custom beans declared using a traditional syntax.
- *UserDetailsService* - closely related to authentication providers, but often also required by other beans.

We'll see how to configure these in the following sections.

18.2. Getting Started with Security Namespace Configuration

In this section, we'll look at how you can build up a namespace configuration to use some of the main features of the framework. Let's assume you initially want to get up and running as quickly as possible and add authentication support and access control to an existing web application, with a few test logins. Then we'll look at how to change over to authenticating against a database or other security repository. In later sections we'll introduce more advanced namespace configuration options.

18.2.1. web.xml Configuration

The first thing you need to do is add the following filter declaration to your `web.xml` file:

```
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

This provides a hook into the Spring Security web infrastructure. `DelegatingFilterProxy` is a Spring Framework class which delegates to a filter implementation which is defined as a Spring bean in your application context. In this case, the bean is named "springSecurityFilterChain", which is an internal infrastructure bean created by the namespace to handle web security. Note that you should not use this bean name yourself. Once you've added this to your `web.xml`, you're ready to start editing your application context file. Web security services are configured using the `<http>` element.

18.2.2. A Minimal <http> Configuration

All you need to enable web security to begin with is

```
<http>
<intercept-url pattern="/**" access="hasRole('USER')" />
<form-login />
<logout />
</http>
```

Which says that we want all URLs within our application to be secured, requiring the role `ROLE_USER` to access them, we want to log in to the application using a form with username and password, and that we want a logout URL registered which will allow us to log out of the application. `<http>` element is the parent for all web-related namespace functionality. The `<intercept-url>` element defines a `pattern` which is matched against the URLs of incoming requests using an ant path style syntax [11]. You can also use regular-expression matching as an alternative (see the namespace appendix for more details). The `access` attribute defines the access requirements for requests matching the given pattern. With the default configuration, this is typically a comma-separated list of roles, one of which a user must have to be allowed to make the request. The prefix "ROLE_" is a marker which indicates that a simple comparison with the user's authorities should be made. In other words, a normal role-based check should be used. Access-control in Spring Security is not limited to the use of simple roles (hence the use of the prefix to differentiate between different types of security attributes). We'll see later how the interpretation can vary [12]. In Spring Security 3.0, the attribute can also be populated with an EL expression.

You can use multiple `<intercept-url>` elements to define different access requirements for different sets of URLs, but they will be evaluated in the order listed and the first match will be used. So you must put the most specific matches at the top. You can also add a `method` attribute to limit the match to a particular HTTP method (`GET`, `POST`, `PUT` etc.).

To add some users, you can define a set of test data directly in the namespace:

```
<authentication-manager>
<authentication-provider>
    <user-service>
    <!-- Password is prefixed with {noop} to indicate to DelegatingPasswordEncoder
that
    NoOpPasswordEncoder should be used. This is not safe for production, but makes
reading
    in samples easier. Normally passwords should be hashed using BCrypt -->
    <user name="jimi" password="{noop}jimispassword" authorities="ROLE_USER,
ROLE_ADMIN" />
    <user name="bob" password="{noop}bobspassword" authorities="ROLE_USER" />
    </user-service>
</authentication-provider>
</authentication-manager>
```

This is an example of a secure way of storing the same passwords. The password is prefixed with `{bcrypt}` to instruct `DelegatingPasswordEncoder`, which supports any configured `PasswordEncoder` for matching, that the passwords are hashed using BCrypt:

```
<authentication-manager>
<authentication-provider>
    <user-service>
    <user name="jimi"
password="{bcrypt}$2a$10$ddEWZUl8aU0GdZPPpy7wbu82dvEw/pBpbRvDQRqA41y6mK1CoH00m"
            authorities="ROLE_USER, ROLE_ADMIN" />
    <user name="bob"
password="{bcrypt}$2a$10$/elFpMBnAYYig6KRR5bvOOYeZr1ie1hSogJryg9qDlhza4oCw1Qka"
            authorities="ROLE_USER" />
    <user name="jimi" password="{noop}jimispassword" authorities="ROLE_USER,
ROLE_ADMIN" />
    <user name="bob" password="{noop}bobspassword" authorities="ROLE_USER" />
    </user-service>
</authentication-provider>
</authentication-manager>
```

If you are familiar with pre-namespace versions of the framework, you can probably already guess roughly what's going on here. The `<http>` element is responsible for creating a `FilterChainProxy` and the filter beans which it uses. Common problems like incorrect filter ordering are no longer an issue as the filter positions are predefined.

The `<authentication-provider>` element creates a `DaoAuthenticationProvider` bean and the `<user-service>` element creates an `InMemoryDaoImpl`. All `authentication-provider` elements must be children of the `<authentication-manager>` element, which creates a `ProviderManager` and registers the authentication providers with it. You can find more detailed information on the beans that are created in the namespace appendix. It's worth cross-checking this if you want to start understanding what the important classes in the framework are and how they are used, particularly if you want to customise things later.

The configuration above defines two users, their passwords and their roles within the application (which will be used for access control). It is also possible to load user information from a standard properties file using the `properties` attribute on `user-service`. See the section on in-memory authentication for more details on the file format. Using the `<authentication-provider>` element means that the user information will be used by the authentication manager to process authentication requests. You can have multiple `<authentication-provider>` elements to define different authentication sources and each will be consulted in turn.

At this point you should be able to start up your application and you will be required to log in to proceed. Try it out, or try experimenting with the "tutorial" sample application that comes with the project.

Setting a Default Post-Login Destination

If a form login isn't prompted by an attempt to access a protected resource, the `default-target-url` option comes into play. This is the URL the user will be taken to after successfully logging in, and defaults to "/". You can also configure things so that the user *always* ends up at this page (regardless of whether the login was "on-demand" or they explicitly chose to log in) by setting the `always-use-default-target` attribute to "true". This is useful if your application always requires that the user starts at a "home" page, for example:

```
<http pattern="/login.htm*" security="none"/>
<http use-expressions="false">
<intercept-url pattern='/**' access='ROLE_USER' />
<form-login login-page='/login.htm' default-target-url='/home.htm'
        always-use-default-target='true' />
</http>
```

For even more control over the destination, you can use the `authentication-success-handler-ref` attribute as an alternative to `default-target-url`. The referenced bean should be an instance of `AuthenticationSuccessHandler`.

18.3. Advanced Web Features

18.3.1. Adding in Your Own Filters

If you've used Spring Security before, you'll know that the framework maintains a chain of filters in order to apply its services. You may want to add your own filters to the stack at particular locations or use a Spring Security filter for which there isn't currently a namespace configuration option (CAS, for example). Or you might want to use a customized version of a standard namespace filter, such as the `UsernamePasswordAuthenticationFilter` which is created by the `<form-login>` element, taking advantage of some of the extra configuration options which are available by using the bean explicitly. How can you do this with namespace configuration, since the filter chain is not directly exposed?

The order of the filters is always strictly enforced when using the namespace. When the application context is being created, the filter beans are sorted by the namespace handling code and the standard Spring Security filters each have an alias in the namespace and a well-known position.

In previous versions, the sorting took place after the filter instances had been created, during post-processing of the application context. In version 3.0+ the sorting is now done at the bean metadata level, before the classes have been instantiated. This has implications for how you add your own filters to the stack as the entire filter list must be known during the parsing of the `<http>` element, so the syntax has changed slightly in 3.0.

The filters, aliases and namespace elements/attributes which create the filters are shown in Standard Filter Aliases and Ordering. The filters are listed in the order in which they occur in the filter chain.

Table 11. Standard Filter Aliases and Ordering

Alias	Filter Class	Namespace Element or Attribute
CHANNEL_FILTER	`ChannelProcessingFilter`	`http/intercept-url@requires-channel`
SECURITY_CONTEXT_FILTER	`SecurityContextPersistenceFilter`	`http`
CONCURRENT_SESSION_FILTER	`ConcurrentSessionFilter`	`session-management/concurrency-control`
HEADERS_FILTER	`HeaderWriterFilter`	`http/headers`
CSRF_FILTER	`CsrfFilter`	`http/csrf`
LOGOUT_FILTER	`LogoutFilter`	`http/logout`
X509_FILTER	`X509AuthenticationFilter`	`http/x509`
PRE_AUTH_FILTER	`AbstractPreAuthenticatedProcessingFilter` Subclasses	N/A
CAS_FILTER	`CasAuthenticationFilter`	N/A

Alias	Filter Class	Namespace Element or Attribute
FORM_LOGIN_FILTER	UsernamePasswordAuthenticationFilter	http/form-login
BASIC_AUTH_FILTER	BasicAuthenticationFilter	http/http-basic
SERVLET_API_SUPPORT_FILTER	SecurityContextHolderAwareRequestFilter	http/@servlet-api-provision
JAAS_API_SUPPORT_FILTER	JaasApiIntegrationFilter	http/@jaas-api-provision
REMEMBER_ME_FILTER	RememberMeAuthenticationFilter	http/remember-me
ANONYMOUS_FILTER	AnonymousAuthenticationFilter	http/anonymous
SESSION_MANAGEMENT_FILTER	SessionManagementFilter	session-management
EXCEPTION_TRANSLATION_FILTER	ExceptionTranslationFilter	http
FILTER_SECURITY_INTERCEPTOR	FilterSecurityInterceptor	http
SWITCH_USER_FILTER	SwitchUserFilter	N/A

You can add your own filter to the stack, using the `custom-filter` element and one of these names to specify the position your filter should appear at:

```
<http>
<custom-filter position="FORM_LOGIN_FILTER" ref="myFilter" />
</http>

<beans:bean id="myFilter" class="com.mycompany.MySpecialAuthenticationFilter"/>
```

You can also use the `after` or `before` attributes if you want your filter to be inserted before or after another filter in the stack. The names "FIRST" and "LAST" can be used with the `position` attribute to indicate that you want your filter to appear before or after the entire stack, respectively.

Avoiding filter position conflicts

If you are inserting a custom filter which may occupy the same position as one of the standard filters created by the namespace then it's important that you don't include the namespace versions by mistake. Remove any elements which create filters whose functionality you want to replace.

Note that you can't replace filters which are created by the use of the `<http>` element itself - `SecurityContextPersistenceFilter`, `ExceptionTranslationFilter` or `FilterSecurityInterceptor`. Some other filters are added by default, but you can disable them. An `AnonymousAuthenticationFilter` is added by default and unless you have session-fixation protection disabled, a `SessionManagementFilter` will also be added to the filter chain.

If you're replacing a namespace filter which requires an authentication entry point (i.e. where the authentication process is triggered by an attempt by an unauthenticated user to access to a secured resource), you will need to add a custom entry point bean too.

18.4. Method Security

From version 2.0 onwards Spring Security has improved support substantially for adding security to your service layer methods. It provides support for JSR-250 annotation security as well as the framework's original `@Secured` annotation. From 3.0 you can also make use of new expression-based annotations. You can apply security to a single bean, using the `intercept-methods` element to decorate the bean declaration, or you can secure multiple beans across the entire service layer using the AspectJ style pointcuts.

18.5. The Default AccessDecisionManager

This section assumes you have some knowledge of the underlying architecture for access-control within Spring Security. If you don't you can skip it and come back to it later, as this section is only really relevant for people who need to do some customization in order to use more than simple role-based security.

When you use a namespace configuration, a default instance of `AccessDecisionManager` is automatically registered for you and will be used for making access decisions for method invocations and web URL access, based on the access attributes you specify in your `intercept-url` and `protect-pointcut` declarations (and in annotations if you are using annotation secured methods).

The default strategy is to use an `AffirmativeBased AccessDecisionManager` with a `RoleVoter` and an `AuthenticatedVoter`. You can find out more about these in the chapter on authorization.

18.5.1. Customizing the AccessDecisionManager

If you need to use a more complicated access control strategy then it is easy to set an alternative for both method and web security.

For method security, you do this by setting the `access-decision-manager-ref` attribute on `global-method-security` to the `id` of the appropriate `AccessDecisionManager` bean in the application context:

```
<global-method-security access-decision-manager-ref="myAccessDecisionManagerBean">
...
</global-method-security>
```

The syntax for web security is the same, but on the `http` element:

```
<http access-decision-manager-ref="myAccessDecisionManagerBean">
...
</http>
```

[10] You can find out more about the use of the `ldap-server` element in the chapter on LDAP Authentication.

[11] See the section on `HttpFirewall` for more details on how matches are actually performed.

[12] The interpretation of the comma-separated values in the `access` attribute depends on the implementation of the AccessDecisionManager which is used.

Chapter 19. Testing

This section describes the testing support provided by Spring Security.

To use the Spring Security test support, you must include `spring-security-test-5.6.0-M1.jar` as a dependency of your project.

19.1. Testing Method Security

This section demonstrates how to use Spring Security's Test support to test method based security. We first introduce a `MessageService` that requires the user to be authenticated in order to access it.

Java

```
public class HelloMessageService implements MessageService {

    @PreAuthorize("authenticated")
    public String getMessage() {
        Authentication authentication = SecurityContextHolder.getContext()
            .getAuthentication();
        return "Hello " + authentication;
    }
}
```

Kotlin

```
class HelloMessageService : MessageService {
    @PreAuthorize("authenticated")
    fun getMessage(): String {
        val authentication: Authentication =
SecurityContextHolder.getContext().authentication
        return "Hello $authentication"
    }
}
```

The result of `getMessage` is a String saying "Hello" to the current Spring Security `Authentication`. An example of the output is displayed below.

```
Hello
org.springframework.security.authentication.UsernamePasswordAuthenticationToken@ca2536
0: Principal: org.springframework.security.core.userdetails.User@36ebcb: Username:
user; Password: [PROTECTED]; Enabled: true; AccountNonExpired: true;
credentialsNonExpired: true; AccountNonLocked: true; Granted Authorities: ROLE_USER;
Credentials: [PROTECTED]; Authenticated: true; Details: null; Granted Authorities:
ROLE_USER
```

19.1.1. Security Test Setup

Before we can use Spring Security Test support, we must perform some setup. An example can be seen below:

Java

```
@RunWith(SpringJUnit4ClassRunner.class) ①
@ContextConfiguration ②
public class WithMockUserTests {
```

Kotlin

```
@RunWith(SpringJUnit4ClassRunner::class)
@ContextConfiguration
class WithMockUserTests {
```

This is a basic example of how to setup Spring Security Test. The highlights are:

① `@RunWith` instructs the spring-test module that it should create an `ApplicationContext`. This is no different than using the existing Spring Test support. For additional information, refer to the Spring Reference

② `@ContextConfiguration` instructs the spring-test the configuration to use to create the `ApplicationContext`. Since no configuration is specified, the default configuration locations will be tried. This is no different than using the existing Spring Test support. For additional information, refer to the Spring Reference

Spring Security hooks into Spring Test support using the `WithSecurityContextTestExecutionListener` which will ensure our tests are ran with the correct user. It does this by populating the `SecurityContextHolder` prior to running our tests. If you are using reactive method security, you will also need `ReactorContextTestExecutionListener` which populates `ReactiveSecurityContextHolder`. After the test is done, it will clear out the `SecurityContextHolder`. If you only need Spring Security related support, you can replace `@ContextConfiguration` with `@SecurityTestExecutionListeners`.

Remember we added the `@PreAuthorize` annotation to our `HelloMessageService` and so it requires an authenticated user to invoke it. If we ran the following test, we would expect the following test will pass:

Java

```
@Test(expected = AuthenticationCredentialsNotFoundException.class)
public void getMessageUnauthenticated() {
    messageService.getMessage();
}
```

Kotlin

```
@Test(expected = AuthenticationCredentialsNotFoundException::class)
fun getMessageUnauthenticated() {
    messageService.getMessage()
}
```

19.1.2. @WithMockUser

The question is "How could we most easily run the test as a specific user?" The answer is to use `@WithMockUser`. The following test will be run as a user with the username "user", the password "password", and the roles "ROLE_USER".

Java

```
@Test
@WithMockUser
public void getMessageWithMockUser() {
String message = messageService.getMessage();
...
}
```

Kotlin

```
@Test
@WithMockUser
fun getMessageWithMockUser() {
    val message: String = messageService.getMessage()
    // ...
}
```

Specifically the following is true:

- The user with the username "user" does not have to exist since we are mocking the user
- The `Authentication` that is populated in the `SecurityContext` is of type `UsernamePasswordAuthenticationToken`
- The principal on the `Authentication` is Spring Security's `User` object

- The `User` will have the username of "user", the password "password", and a single `GrantedAuthority` named "ROLE_USER" is used.

Our example is nice because we are able to leverage a lot of defaults. What if we wanted to run the test with a different username? The following test would run with the username "customUser". Again, the user does not need to actually exist.

Java

```
@Test
@WithMockUser("customUsername")
public void getMessageWithMockUserCustomUsername() {
    String message = messageService.getMessage();
...
}
```

Kotlin

```
@Test
@WithMockUser("customUsername")
fun getMessageWithMockUserCustomUsername() {
    val message: String = messageService.getMessage()
    // ...
}
```

We can also easily customize the roles. For example, this test will be invoked with the username "admin" and the roles "ROLE_USER" and "ROLE_ADMIN".

Java

```
@Test
@WithMockUser(username="admin",roles={"USER","ADMIN"})
public void getMessageWithMockUserCustomUser() {
    String message = messageService.getMessage();
    ...
}
```

Kotlin

```
@Test
@WithMockUser(username="admin",roles=["USER","ADMIN"])
fun getMessageWithMockUserCustomUser() {
    val message: String = messageService.getMessage()
    // ...
}
```

If we do not want the value to automatically be prefixed with ROLE_ we can leverage the authorities attribute. For example, this test will be invoked with the username "admin" and the authorities "USER" and "ADMIN".

Java

```
@Test
@WithMockUser(username = "admin", authorities = { "ADMIN", "USER" })
public void getMessageWithMockUserCustomAuthorities() {
    String message = messageService.getMessage();
    ...
}
```

Kotlin

```
@Test
@WithMockUser(username = "admin", authorities = ["ADMIN", "USER"])
fun getMessageWithMockUserCustomUsername() {
    val message: String = messageService.getMessage()
    // ...
}
```

Of course it can be a bit tedious placing the annotation on every test method. Instead, we can place the annotation at the class level and every test will use the specified user. For example, the following would run every test with a user with the username "admin", the password "password", and the roles "ROLE_USER" and "ROLE_ADMIN".

Java

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WithMockUser(username="admin",roles={"USER","ADMIN"})
public class WithMockUserTests {
```

Kotlin

```
@RunWith(SpringJUnit4ClassRunner::class)
@ContextConfiguration
@WithMockUser(username="admin",roles=["USER","ADMIN"])
class WithMockUserTests {
```

If you are using JUnit 5's `@Nested` test support, you can also place the annotation on the enclosing class to apply to all nested classes. For example, the following would run every test with a user with the username "admin", the password "password", and the roles "ROLE_USER" and "ROLE_ADMIN" for both test methods.

Java

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration
@WithMockUser(username="admin",roles={"USER","ADMIN"})
public class WithMockUserTests {

    @Nested
    public class TestSuite1 {
        // ... all test methods use admin user
    }

    @Nested
    public class TestSuite2 {
        // ... all test methods use admin user
    }
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
@ContextConfiguration
@WithMockUser(username = "admin", roles = ["USER", "ADMIN"])
class WithMockUserTests {
    @Nested
    inner class TestSuite1 { // ... all test methods use admin user
    }

    @Nested
    inner class TestSuite2 { // ... all test methods use admin user
    }
}
```

By default the `SecurityContext` is set during the `TestExecutionListener.beforeTestMethod` event. This is the equivalent of happening before JUnit's `@Before`. You can change this to happen during the `TestExecutionListener.beforeTestExecution` event which is after JUnit's `@Before` but before the test method is invoked.

```
@WithMockUser(setupBefore = TestExecutionEvent.TEST_EXECUTION)
```

19.1.3. @WithAnonymousUser

Using `@WithAnonymousUser` allows running as an anonymous user. This is especially convenient when you wish to run most of your tests with a specific user, but want to run a few tests as an anonymous user. For example, the following will run withMockUser1 and withMockUser2 using @WithMockUser and anonymous as an anonymous user.

Java

```
@RunWith(SpringJUnit4ClassRunner.class)
@WithMockUser
public class WithUserClassLevelAuthenticationTests {

    @Test
    public void withMockUser1() {
    }

    @Test
    public void withMockUser2() {
    }

    @Test
    @WithAnonymousUser
    public void anonymous() throws Exception {
        // override default to run as anonymous user
    }
}
```

Kotlin

```
@RunWith(SpringJUnit4ClassRunner::class)
@WithMockUser
class WithUserClassLevelAuthenticationTests {
    @Test
    fun withMockUser1() {
    }

    @Test
    fun withMockUser2() {
    }

    @Test
    @WithAnonymousUser
    fun anonymous() {
        // override default to run as anonymous user
    }
}
```

By default the `SecurityContext` is set during the `TestExecutionListener.beforeTestMethod` event. This is the equivalent of happening before JUnit's `@Before`. You can change this to happen during the `TestExecutionListener.beforeTestExecution` event which is after JUnit's `@Before` but before the test method is invoked.

```
@WithAnonymousUser(setupBefore = TestExecutionEvent.TEST_EXECUTION)
```

19.1.4. @WithUserDetails

While `@WithMockUser` is a very convenient way to get started, it may not work in all instances. For example, it is common for applications to expect that the `Authentication` principal be of a specific type. This is done so that the application can refer to the principal as the custom type and reduce coupling on Spring Security.

The custom principal is often times returned by a custom `UserDetailsService` that returns an object that implements both `UserDetails` and the custom type. For situations like this, it is useful to create the test user using the custom `UserDetailsService`. That is exactly what `@WithUserDetails` does.

Assuming we have a `UserDetailsService` exposed as a bean, the following test will be invoked with an `Authentication` of type `UsernamePasswordAuthenticationToken` and a principal that is returned from the `UserDetailsService` with the username of "user".

Java

```
@Test
@WithUserDetails
public void getMessageWithUserDetails() {
    String message = messageService.getMessage();
    ...
}
```

Kotlin

```
@Test
@WithUserDetails
fun getMessageWithUserDetails() {
    val message: String = messageService.getMessage()
    // ...
}
```

We can also customize the username used to lookup the user from our `UserDetailsService`. For example, this test would be run with a principal that is returned from the `UserDetailsService` with the username of "customUsername".

Java

```
@Test
@WithUserDetails("customUsername")
public void getMessageWithUserDetailsCustomUsername() {
    String message = messageService.getMessage();
    ...
}
```

Kotlin

```
@Test
@WithUserDetails("customUsername")
fun getMessageWithUserDetailsCustomUsername() {
    val message: String = messageService.getMessage()
    // ...
}
```

We can also provide an explicit bean name to look up the `UserDetailsService`. For example, this test would look up the username of "customUsername" using the `UserDetailsService` with the bean name "myUserDetailsService".

Java

```
@Test
@WithUserDetails(value="customUsername",
userDetailsServiceBeanName="myUserDetailsService")
public void getMessageWithUserDetailsServiceBeanName() {
    String message = messageService.getMessage();
    ...
}
```

Kotlin

```
@Test
@WithUserDetails(value="customUsername",
userDetailsServiceBeanName="myUserDetailsService")
fun getMessageWithUserDetailsServiceBeanName() {
    val message: String = messageService.getMessage()
    // ...
}
```

Like `@WithMockUser` we can also place our annotation at the class level so that every test uses the same user. However unlike `@WithMockUser`, `@WithUserDetails` requires the user to exist.

By default the `SecurityContext` is set during the `TestExecutionListener.beforeTestMethod` event. This is the equivalent of happening before JUnit's `@Before`. You can change this to happen during the `TestExecutionListener.beforeTestExecution` event which is after JUnit's `@Before` but before the test method is invoked.

```
@WithUserDetails(setupBefore = TestExecutionEvent.TEST_EXECUTION)
```

19.1.5. @WithSecurityContext

We have seen that `@WithMockUser` is an excellent choice if we are not using a custom `Authentication` principal. Next we discovered that `@WithUserDetails` would allow us to use a custom `UserDetailsService` to create our `Authentication` principal but required the user to exist. We will now see an option that allows the most flexibility.

We can create our own annotation that uses the `@WithSecurityContext` to create any `SecurityContext` we want. For example, we might create an annotation named `@WithMockCustomUser` as shown below:

Java

```
@Retention(RetentionPolicy.RUNTIME)
@WithSecurityContext(factory = WithMockCustomUserSecurityContextFactory.class)
public @interface WithMockCustomUser {

    String username() default "rob";

    String name() default "Rob Winch";
}
```

Kotlin

```
@Retention(AnnotationRetention.RUNTIME)
@WithSecurityContext(factory = WithMockCustomUserSecurityContextFactory::class)
annotation class WithMockCustomUser(val username: String = "rob", val name: String
= "Rob Winch")
```

You can see that `@WithMockCustomUser` is annotated with the `@WithSecurityContext` annotation. This is what signals to Spring Security Test support that we intend to create a `SecurityContext` for the test. The `@WithSecurityContext` annotation requires we specify a `SecurityContextFactory` that will create a new `SecurityContext` given our `@WithMockCustomUser` annotation. You can find our `WithMockCustomUserSecurityContextFactory` implementation below:

Java

```
public class WithMockCustomUserSecurityContextFactory
    implements WithSecurityContextFactory<WithMockCustomUser> {
    @Override
    public SecurityContext createSecurityContext(WithMockCustomUser customUser) {
        SecurityContext context = SecurityContextHolder.createEmptyContext();

        CustomUserDetails principal =
            new CustomUserDetails(customUser.name(), customUser.username());
        Authentication auth =
            new UsernamePasswordAuthenticationToken(principal, "password",
principal.getAuthorities());
        context.setAuthentication(auth);
        return context;
    }
}
```

Kotlin

```
class WithMockCustomUserSecurityContextFactory :
WithSecurityContextFactory<WithMockCustomUser> {
    override fun createSecurityContext(customUser: WithMockCustomUser):
SecurityContext {
        val context = SecurityContextHolder.createEmptyContext()
        val principal = CustomUserDetails(customUser.name, customUser.username)
        val auth: Authentication =
            UsernamePasswordAuthenticationToken(principal, "password",
principal.authorities)
        context.authentication = auth
        return context
    }
}
```

We can now annotate a test class or a test method with our new annotation and Spring Security's `WithSecurityContextTestExecutionListener` will ensure that our `SecurityContext` is populated appropriately.

When creating your own `WithSecurityContextFactory` implementations, it is nice to know that they can be annotated with standard Spring annotations. For example, the `WithUserDetailsSecurityContextFactory` uses the `@Autowired` annotation to acquire the `UserDetailsService`:

Java

```
final class WithUserDetailsSecurityContextFactory
    implements WithSecurityContextFactory<WithUserDetails> {

    private UserDetailsService userDetailsService;

    @Autowired
    public WithUserDetailsSecurityContextFactory(UserDetailsService
userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    public SecurityContext createSecurityContext(WithUserDetails withUser) {
        String username = withUser.value();
        Assert.hasLength(username, "value() must be non-empty String");
        UserDetails principal = userDetailsService.loadUserByUsername(username);
        Authentication authentication = new
UsernamePasswordAuthenticationToken(principal, principal.getPassword(),
principal.getAuthorities());
        SecurityContext context = SecurityContextHolder.createEmptyContext();
        context.setAuthentication(authentication);
        return context;
    }
}
```

Kotlin

```
class WithUserDetailsSecurityContextFactory @Autowired constructor(private val
userDetailsService: UserDetailsService) :
    WithSecurityContextFactory<WithUserDetails> {
    override fun createSecurityContext(withUser: WithUserDetails): SecurityContext
{
        val username: String = withUser.value
        Assert.hasLength(username, "value() must be non-empty String")
        val principal = userDetailsService.loadUserByUsername(username)
        val authentication: Authentication =
            UsernamePasswordAuthenticationToken(principal, principal.password,
principal.authorities)
        val context = SecurityContextHolder.createEmptyContext()
        context.authentication = authentication
        return context
    }
}
```

By default the `SecurityContext` is set during the `TestExecutionListener.beforeTestMethod` event. This is the equivalent of happening before JUnit's `@Before`. You can change this to happen during the `TestExecutionListener.beforeTestExecution` event which is after JUnit's `@Before` but before the test

method is invoked.

```
@WithSecurityContext(setupBefore = TestExecutionEvent.TEST_EXECUTION)
```

19.1.6. Test Meta Annotations

If you reuse the same user within your tests often, it is not ideal to have to repeatedly specify the attributes. For example, if there are many tests related to an administrative user with the username "admin" and the roles `ROLE_USER` and `ROLE_ADMIN` you would have to write:

Java

```
@WithMockUser(username="admin",roles={"USER","ADMIN"})
```

Kotlin

```
@WithMockUser(username="admin",roles=["USER","ADMIN"])
```

Rather than repeating this everywhere, we can use a meta annotation. For example, we could create a meta annotation named `WithMockAdmin`:

Java

```
@Retention(RetentionPolicy.RUNTIME)
@WithMockUser(value="rob",roles="ADMIN")
public @interface WithMockAdmin { }
```

Kotlin

```
@Retention(AnnotationRetention.RUNTIME)
@WithMockUser(value = "rob", roles = ["ADMIN"])
annotation class WithMockAdmin
```

Now we can use `@WithMockAdmin` in the same way as the more verbose `@WithMockUser`.

Meta annotations work with any of the testing annotations described above. For example, this means we could create a meta annotation for `@WithUserDetails("admin")` as well.

19.2. Spring MVC Test Integration

Spring Security provides comprehensive integration with Spring MVC Test

19.2.1. Setting Up MockMvc and Spring Security

In order to use Spring Security with Spring MVC Test it is necessary to add the Spring Security `FilterChainProxy` as a `Filter`. It is also necessary to add Spring Security's `TestSecurityContextHolderPostProcessor` to support Running as a User in Spring MVC Test with Annotations. This can be done using Spring Security's `SecurityMockMvcConfigurers.springSecurity()`. For example:

Spring Security's testing support requires spring-test-4.1.3.RELEASE or greater.

Java

```
import static
org.springframework.security.test.web.servlet.setup.SecurityMockMvcConfigurers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = SecurityConfig.class)
@WebAppConfiguration
public class CsrfShowcaseTests {

    @Autowired
    private WebApplicationContext context;

    private MockMvc mvc;

    @Before
    public void setup() {
        mvc = MockMvcBuilders
                .webAppContextSetup(context)
                .apply(springSecurity()) ①
                .build();
    }

...
```

Kotlin

```
@RunWith(SpringJUnit4ClassRunner::class)
@ContextConfiguration(classes = [SecurityConfig::class])
@WebAppConfiguration
class CsrfShowcaseTests {

    @Autowired
    private lateinit var context: WebApplicationContext

    private var mvc: MockMvc? = null

    @Before
    fun setup() {
        mvc = MockMvcBuilders
            .webAppContextSetup(context)
            .apply<DefaultMockMvcBuilder>(springSecurity()) ①
            .build()
    }
// ...
```

① `SecurityMockMvcConfigurers.springSecurity()` will perform all of the initial setup we need to integrate Spring Security with Spring MVC Test

19.2.2. SecurityMockMvcRequestPostProcessors

Spring MVC Test provides a convenient interface called a `RequestPostProcessor` that can be used to modify a request. Spring Security provides a number of `RequestPostProcessor` implementations that make testing easier. In order to use Spring Security's `RequestPostProcessor` implementations ensure the following static import is used:

Java

```
import static
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostPr
ocessors.*;
```

Kotlin

```
import
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostPr
ocessors.*
```

Testing with CSRF Protection

When testing any non-safe HTTP methods and using Spring Security's CSRF protection, you must be sure to include a valid CSRF Token in the request. To specify a valid CSRF token as a request parameter using the following:

Java

```
mvc
    .perform(post("/").with(csrf()))
```

Kotlin

```
mvc.post("/") {
    with(csrf())
}
```

If you like you can include CSRF token in the header instead:

Java

```
mvc
    .perform(post("/").with(csrf().asHeader()))
```

Kotlin

```
mvc.post("/") {
    with(csrf().asHeader())
}
```

You can also test providing an invalid CSRF token using the following:

Java

```
mvc
    .perform(post("/").with(csrf().useInvalidToken()))
```

Kotlin

```
mvc.post("/") {
    with(csrf().useInvalidToken())
}
```

Running a Test as a User in Spring MVC Test

It is often desirable to run tests as a specific user. There are two simple ways of populating the user:

- Running as a User in Spring MVC Test with RequestPostProcessor
- Running as a User in Spring MVC Test with Annotations

Running as a User in Spring MVC Test with RequestPostProcessor

There are a number of options available to associate a user to the current `HttpServletRequest`. For example, the following will run as a user (which does not need to exist) with the username "user", the password "password", and the role "ROLE_USER":

The support works by associating the user to the `HttpServletRequest`. To associate the request to the `SecurityContextHolder` you need to ensure that the `SecurityContextPersistenceFilter` is associated with the `MockMvc` instance. A few ways to do this are:

- Invoking apply(springSecurity())
- Adding Spring Security's `FilterChainProxy` to `MockMvc`
- Manually adding `SecurityContextPersistenceFilter` to the `MockMvc` instance may make sense when using `MockMvcBuilders.standaloneSetup`

Java

```
mvc
    .perform(get("/").with(user("user")))
```

Kotlin

```
mvc.get("/") {
    with(user("user"))
}
```

You can easily make customizations. For example, the following will run as a user (which does not need to exist) with the username "admin", the password "pass", and the roles "ROLE_USER" and "ROLE_ADMIN".

Java

```
mvc

.perform(get("/admin").with(user("admin").password("pass").roles("USER","ADMIN")))
```

Kotlin

```
mvc.get("/admin") {
    with(user("admin").password("pass").roles("USER","ADMIN"))
}
```

If you have a custom `UserDetails` that you would like to use, you can easily specify that as well. For example, the following will use the specified `UserDetails` (which does not need to exist) to run with a `UsernamePasswordAuthenticationToken` that has a principal of the specified `UserDetails`:

Java

```
mvc
    .perform(get("/").with(user(userDetails)))
```

Kotlin

```
mvc.get("/") {
    with(user(userDetails))
}
```

You can run as anonymous user using the following:

Java

```
mvc
    .perform(get("/").with(anonymous()))
```

Kotlin

```
mvc.get("/") {
    with(anonymous())
}
```

This is especially useful if you are running with a default user and wish to process a few requests as an anonymous user.

If you want a custom `Authentication` (which does not need to exist) you can do so using the following:

Java

```
mvc
    .perform(get("/").with(authentication(authentication)))
```

Kotlin

```
mvc.get("/") {
    with(authentication(authentication))
}
```

You can even customize the `SecurityContext` using the following:

Java

```
mvc
    .perform(get("/").with(securityContext(securityContext)))
```

Kotlin

```
mvc.get("/") {
    with(securityContext(securityContext))
}
```

We can also ensure to run as a specific user for every request by using `MockMvcBuilders`'s default request. For example, the following will run as a user (which does not need to exist) with the username "admin", the password "password", and the role "ROLE_ADMIN":

Java

```
mvc = MockMvcBuilders
        .webAppContextSetup(context)
        .defaultRequest(get("/").with(user("user").roles("ADMIN")))
        .apply(springSecurity())
        .build();
```

Kotlin

```
mvc = MockMvcBuilders
    .webAppContextSetup(context)

.defaultRequest<DefaultMockMvcBuilder>(get("/").with(user("user").roles("ADMIN")))
    .apply<DefaultMockMvcBuilder>(springSecurity())
    .build()
```

If you find you are using the same user in many of your tests, it is recommended to move the user to a method. For example, you can specify the following in your own class named `CustomSecurityMockMvcRequestPostProcessors`:

Java

```
public static RequestPostProcessor rob() {
    return user("rob").roles("ADMIN");
}
```

Kotlin

```
fun rob(): RequestPostProcessor {
    return user("rob").roles("ADMIN")
}
```

Now you can perform a static import on `SecurityMockMvcRequestPostProcessors` and use that within your tests:

Java

```
import static sample.CustomSecurityMockMvcRequestPostProcessors.*;

...

mvc
    .perform(get("/").with(rob()))
```

Kotlin

```
import sample.CustomSecurityMockMvcRequestPostProcessors.*

//...

mvc.get("/") {
    with(rob())
}
```

Running as a User in Spring MVC Test with Annotations

As an alternative to using a `RequestPostProcessor` to create your user, you can use annotations described in Testing Method Security. For example, the following will run the test with the user with username "user", password "password", and role "ROLE_USER":

Java

```
@Test
@WithMockUser
public void requestProtectedUrlWithUser() throws Exception {
mvc
        .perform(get("/"))
        ...
}
```

Kotlin

```
@Test
@WithMockUser
fun requestProtectedUrlWithUser() {
    mvc
        .get("/")
        // ...
}
```

Alternatively, the following will run the test with the user with username "user", password "password", and role "ROLE_ADMIN":

Java

```
@Test
@WithMockUser(roles="ADMIN")
public void requestProtectedUrlWithUser() throws Exception {
mvc
        .perform(get("/"))
        ...
}
```

Kotlin

```
@Test
@WithMockUser(roles = ["ADMIN"])
fun requestProtectedUrlWithUser() {
    mvc
        .get("/")
        // ...
}
```

Testing HTTP Basic Authentication

While it has always been possible to authenticate with HTTP Basic, it was a bit tedious to remember the header name, format, and encode the values. Now this can be done using Spring Security's `httpBasic RequestPostProcessor`. For example, the snippet below:

Java

```
mvc
    .perform(get("/").with(httpBasic("user","password")))
```

Kotlin

```
mvc.get("/") {
    with(httpBasic("user","password"))
}
```

will attempt to use HTTP Basic to authenticate a user with the username "user" and the password "password" by ensuring the following header is populated on the HTTP Request:

```
Authorization: Basic dXNlcjpwYXNzd29yZA==
```

Testing OAuth 2.0

When it comes to OAuth 2.0, the same principles covered earlier still apply: Ultimately, it depends on what your method under test is expecting to be in the `SecurityContextHolder`.

For example, for a controller that looks like this:

Java

```
@GetMapping("/endpoint")
public String foo(Principal user) {
    return user.getName();
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(user: Principal): String {
    return user.name
}
```

There's nothing OAuth2-specific about it, so you will likely be able to simply use `@WithMockUser` and

be fine.

But, in cases where your controllers are bound to some aspect of Spring Security's OAuth 2.0 support, like the following:

Java

```
@GetMapping("/endpoint")
public String foo(@AuthenticationPrincipal OidcUser user) {
    return user.getIdToken().getSubject();
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(@AuthenticationPrincipal user: OidcUser): String {
    return user.idToken.subject
}
```

then Spring Security's test support can come in handy.

Testing OIDC Login

Testing the method above with Spring MVC Test would require simulating some kind of grant flow with an authorization server. Certainly this would be a daunting task, which is why Spring Security ships with support for removing this boilerplate.

For example, we can tell Spring Security to include a default `OidcUser` using the `SecurityMockMvcRequestPostProcessors#oidcLogin` method, like so:

Java

```
mvc
    .perform(get("/endpoint").with(oidcLogin()));
```

Kotlin

```
mvc.get("/endpoint") {
    with(oidcLogin())
}
```

What this will do is configure the associated `MockHttpServletRequest` with an `OidcUser` that includes a simple `OidcIdToken`, `OidcUserInfo`, and `Collection` of granted authorities.

Specifically, it will include an `OidcIdToken` with a `sub` claim set to `user`:

Java

```
assertThat(user.getIdToken().getClaim("sub")).isEqualTo("user");
```

Kotlin

```
assertThat(user.idToken.getClaim<String>("sub")).isEqualTo("user")
```

an `OidcUserInfo` with no claims set:

Java

```
assertThat(user.getUserInfo().getClaims()).isEmpty();
```

Kotlin

```
assertThat(user.userInfo.claims).isEmpty()
```

and a `Collection` of authorities with just one authority, `SCOPE_read`:

Java

```
assertThat(user.getAuthorities()).hasSize(1);
assertThat(user.getAuthorities()).containsExactly(new
SimpleGrantedAuthority("SCOPE_read"));
```

Kotlin

```
assertThat(user.authorities).hasSize(1)
assertThat(user.authorities).containsExactly(SimpleGrantedAuthority("SCOPE_read"))
```

Spring Security does the necessary work to make sure that the `OidcUser` instance is available for the `@AuthenticationPrincipal` annotation.

Further, it also links that `OidcUser` to a simple instance of `OAuth2AuthorizedClient` that it deposits into an mock `OAuth2AuthorizedClientRepository`. This can be handy if your tests use the `@RegisteredOAuth2AuthorizedClient` annotation..

Configuring Authorities

In many circumstances, your method is protected by filter or method security and needs your `Authentication` to have certain granted authorities to allow the request.

In this case, you can supply what granted authorities you need using the `authorities()` method:

Java

```
mvc
    .perform(get("/endpoint")
        .with(oidcLogin()
            .authorities(new SimpleGrantedAuthority("SCOPE_message:read"))
        )
    );
```

Kotlin

```
mvc.get("/endpoint") {
    with(oidcLogin()
        .authorities(SimpleGrantedAuthority("SCOPE_message:read"))
    )
}
```

Configuring Claims

And while granted authorities are quite common across all of Spring Security, we also have claims in the case of OAuth 2.0.

Let's say, for example, that you've got a `user_id` claim that indicates the user's id in your system. You might access it like so in a controller:

Java

```
@GetMapping("/endpoint")
public String foo(@AuthenticationPrincipal OidcUser oidcUser) {
    String userId = oidcUser.getIdToken().getClaim("user_id");
    // ...
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(@AuthenticationPrincipal oidcUser: OidcUser): String {
    val userId = oidcUser.idToken.getClaim<String>("user_id")
    // ...
}
```

In that case, you'd want to specify that claim with the `idToken()` method:

Java

```
mvc
    .perform(get("/endpoint")
        .with(oidcLogin()
                .idToken(token -> token.claim("user_id", "1234"))
        )
    );
```

Kotlin

```
mvc.get("/endpoint") {
    with(oidcLogin()
        .idToken {
            it.claim("user_id", "1234")
        }
    )
}
```

since `OidcUser` collects its claims from `OidcIdToken`.

Additional Configurations

There are additional methods, too, for further configuring the authentication; it simply depends on what data your controller expects:

- `userInfo(OidcUserInfo.Builder)` - For configuring the `OidcUserInfo` instance
- `clientRegistration(ClientRegistration)` - For configuring the associated `OAuth2AuthorizedClient` with a given `ClientRegistration`
- `oidcUser(OidcUser)` - For configuring the complete `OidcUser` instance

That last one is handy if you: 1. Have your own implementation of `OidcUser`, or 2. Need to change the name attribute

For example, let's say that your authorization server sends the principal name in the `user_name` claim instead of the `sub` claim. In that case, you can configure an `OidcUser` by hand:

Java

```
OidcUser oidcUser = new DefaultOidcUser(
        AuthorityUtils.createAuthorityList("SCOPE_message:read"),
        OidcIdToken.withTokenValue("id-token").claim("user_name",
"foo_user").build(),
        "user_name");

mvc
    .perform(get("/endpoint")
        .with(oidcLogin().oidcUser(oidcUser))
    );
```

Kotlin

```
val oidcUser: OidcUser = DefaultOidcUser(
    AuthorityUtils.createAuthorityList("SCOPE_message:read"),
    OidcIdToken.withTokenValue("id-token").claim("user_name", "foo_user").build(),
    "user_name"
)

mvc.get("/endpoint") {
    with(oidcLogin().oidcUser(oidcUser))
}
```

Testing OAuth 2.0 Login

As with testing OIDC login, testing OAuth 2.0 Login presents a similar challenge of mocking a grant flow. And because of that, Spring Security also has test support for non-OIDC use cases.

Let's say that we've got a controller that gets the logged-in user as an `OAuth2User`:

Java

```
@GetMapping("/endpoint")
public String foo(@AuthenticationPrincipal OAuth2User oauth2User) {
    return oauth2User.getAttribute("sub");
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(@AuthenticationPrincipal oauth2User: OAuth2User): String? {
    return oauth2User.getAttribute("sub")
}
```

In that case, we can tell Spring Security to include a default `OAuth2User` using the `SecurityMockMvcRequestPostProcessors#oauth2User` method, like so:

Java

```
mvc
    .perform(get("/endpoint").with(oauth2Login()));
```

Kotlin

```
mvc.get("/endpoint") {
    with(oauth2Login())
}
```

What this will do is configure the associated `MockHttpServletRequest` with an `OAuth2User` that includes a simple `Map` of attributes and `Collection` of granted authorities.

Specifically, it will include a `Map` with a key/value pair of `sub`/`user`:

Java

```
assertThat((String) user.getAttribute("sub")).isEqualTo("user");
```

Kotlin

```
assertThat(user.getAttribute<String>("sub")).isEqualTo("user")
```

and a `Collection` of authorities with just one authority, `SCOPE_read`:

Java

```
assertThat(user.getAuthorities()).hasSize(1);
assertThat(user.getAuthorities()).containsExactly(new
SimpleGrantedAuthority("SCOPE_read"));
```

Kotlin

```
assertThat(user.authorities).hasSize(1)
assertThat(user.authorities).containsExactly(SimpleGrantedAuthority("SCOPE_read"))
```

Spring Security does the necessary work to make sure that the `OAuth2User` instance is available for the `@AuthenticationPrincipal` annotation.

Further, it also links that `OAuth2User` to a simple instance of `OAuth2AuthorizedClient` that it deposits in a mock `OAuth2AuthorizedClientRepository`. This can be handy if your tests use the `@RegisteredOAuth2AuthorizedClient` annotation.

Configuring Authorities

In many circumstances, your method is protected by filter or method security and needs your `Authentication` to have certain granted authorities to allow the request.

In this case, you can supply what granted authorities you need using the `authorities()` method:

Java

```
mvc
    .perform(get("/endpoint")
        .with(oauth2Login()
            .authorities(new SimpleGrantedAuthority("SCOPE_message:read"))
        )
    );
```

Kotlin

```
mvc.get("/endpoint") {
    with(oauth2Login()
        .authorities(SimpleGrantedAuthority("SCOPE_message:read"))
    )
}
```

Configuring Claims

And while granted authorities are quite common across all of Spring Security, we also have claims in the case of OAuth 2.0.

Let's say, for example, that you've got a `user_id` attribute that indicates the user's id in your system. You might access it like so in a controller:

Java

```
@GetMapping("/endpoint")
public String foo(@AuthenticationPrincipal OAuth2User oauth2User) {
    String userId = oauth2User.getAttribute("user_id");
    // ...
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(@AuthenticationPrincipal oauth2User: OAuth2User): String {
    val userId = oauth2User.getAttribute<String>("user_id")
    // ...
}
```

In that case, you'd want to specify that attribute with the `attributes()` method:

Java

```
mvc
    .perform(get("/endpoint")
        .with(oauth2Login()
                .attributes(attrs -> attrs.put("user_id", "1234"))
        )
    );
```

Kotlin

```
mvc.get("/endpoint") {
    with(oauth2Login()
        .attributes { attrs -> attrs["user_id"] = "1234" }
    )
}
```

Additional Configurations

There are additional methods, too, for further configuring the authentication; it simply depends on what data your controller expects:

- `clientRegistration(ClientRegistration)` - For configuring the associated `OAuth2AuthorizedClient` with a given `ClientRegistration`
- `oauth2User(OAuth2User)` - For configuring the complete `OAuth2User` instance

That last one is handy if you: 1. Have your own implementation of `OAuth2User`, or 2. Need to change

the name attribute

For example, let's say that your authorization server sends the principal name in the `user_name` claim instead of the `sub` claim. In that case, you can configure an `OAuth2User` by hand:

Java

```
OAuth2User oauth2User = new DefaultOAuth2User(
        AuthorityUtils.createAuthorityList("SCOPE_message:read"),
        Collections.singletonMap("user_name", "foo_user"),
        "user_name");

mvc
    .perform(get("/endpoint")
        .with(oauth2Login().oauth2User(oauth2User))
    );
```

Kotlin

```
val oauth2User: OAuth2User = DefaultOAuth2User(
    AuthorityUtils.createAuthorityList("SCOPE_message:read"),
    mapOf(Pair("user_name", "foo_user")),
    "user_name"
)

mvc.get("/endpoint") {
    with(oauth2Login().oauth2User(oauth2User))
}
```

Testing OAuth 2.0 Clients

Independent of how your user authenticates, you may have other tokens and client registrations that are in play for the request you are testing. For example, your controller may be relying on the client credentials grant to get a token that isn't associated with the user at all:

Java

```
@GetMapping("/endpoint")
public String foo(@RegisteredOAuth2AuthorizedClient("my-app")
OAuth2AuthorizedClient authorizedClient) {
    return this.webClient.get()
        .attributes(oauth2AuthorizedClient(authorizedClient))
        .retrieve()
        .bodyToMono(String.class)
        .block();
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(@RegisteredOAuth2AuthorizedClient("my-app") authorizedClient:
OAuth2AuthorizedClient?): String? {
    return this.webClient.get()
        .attributes(oauth2AuthorizedClient(authorizedClient))
        .retrieve()
        .bodyToMono(String::class.java)
        .block()
}
```

Simulating this handshake with the authorization server could be cumbersome. Instead, you can use `SecurityMockMvcRequestPostProcessor#oauth2Client` to add a `OAuth2AuthorizedClient` into a mock `OAuth2AuthorizedClientRepository`:

Java

```
mvc
    .perform(get("/endpoint").with(oauth2Client("my-app")));
```

Kotlin

```
mvc.get("/endpoint") {
    with(
        oauth2Client("my-app")
    )
}
```

What this will do is create an `OAuth2AuthorizedClient` that has a simple `ClientRegistration`, `OAuth2AccessToken`, and resource owner name.

Specifically, it will include a `ClientRegistration` with a client id of "test-client" and client secret of

"test-secret":

Java

```
assertThat(authorizedClient.getClientRegistration().getClientId()).isEqualTo("test
-client");
assertThat(authorizedClient.getClientRegistration().getClientSecret()).isEqualTo("
test-secret");
```

Kotlin

```
assertThat(authorizedClient.clientRegistration.clientId).isEqualTo("test-client")
assertThat(authorizedClient.clientRegistration.clientSecret).isEqualTo("test-
secret")
```

a resource owner name of "user":

Java

```
assertThat(authorizedClient.getPrincipalName()).isEqualTo("user");
```

Kotlin

```
assertThat(authorizedClient.principalName).isEqualTo("user")
```

and an `OAuth2AccessToken` with just one scope, `read`:

Java

```
assertThat(authorizedClient.getAccessToken().getScopes()).hasSize(1);
assertThat(authorizedClient.getAccessToken().getScopes()).containsExactly("read");
```

Kotlin

```
assertThat(authorizedClient.accessToken.scopes).hasSize(1)
assertThat(authorizedClient.accessToken.scopes).containsExactly("read")
```

The client can then be retrieved as normal using `@RegisteredOAuth2AuthorizedClient` in a controller method.

Configuring Scopes

In many circumstances, the OAuth 2.0 access token comes with a set of scopes. If your controller

inspects these, say like so:

Java

```
@GetMapping("/endpoint")
public String foo(@RegisteredOAuth2AuthorizedClient("my-app")
OAuth2AuthorizedClient authorizedClient) {
    Set<String> scopes = authorizedClient.getAccessToken().getScopes();
    if (scopes.contains("message:read")) {
        return this.webClient.get()
            .attributes(oauth2AuthorizedClient(authorizedClient))
            .retrieve()
            .bodyToMono(String.class)
            .block();
    }
    // ...
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(@RegisteredOAuth2AuthorizedClient("my-app") authorizedClient:
OAuth2AuthorizedClient): String? {
    val scopes = authorizedClient.accessToken.scopes
    if (scopes.contains("message:read")) {
        return webClient.get()
            .attributes(oauth2AuthorizedClient(authorizedClient))
            .retrieve()
            .bodyToMono(String::class.java)
            .block()
    }
    // ...
}
```

then you can configure the scope using the `accessToken()` method:

Java

```
mvc
    .perform(get("/endpoint")
        .with(oauth2Client("my-app")
            .accessToken(new OAuth2AccessToken(BEARER, "token", null, null,
Collections.singleton("message:read"))))
        )
    );
```

Kotlin

```
mvc.get("/endpoint") {
    with(oauth2Client("my-app")
            .accessToken(OAuth2AccessToken(BEARER, "token", null, null,
Collections.singleton("message:read")))
    )
}
```

Additional Configurations

There are additional methods, too, for further configuring the authentication; it simply depends on what data your controller expects:

- `principalName(String)` - For configuring the resource owner name
- `clientRegistration(Consumer<ClientRegistration.Builder>)` - For configuring the associated `ClientRegistration`
- `clientRegistration(ClientRegistration)` - For configuring the complete `ClientRegistration`

That last one is handy if you want to use a real `ClientRegistration`

For example, let's say that you are wanting to use one of your app's `ClientRegistration` definitions, as specified in your `application.yml`.

In that case, your test can autowire the `ClientRegistrationRepository` and look up the one your test needs:

Java

```
@Autowired
ClientRegistrationRepository clientRegistrationRepository;

// ...

mvc
    .perform(get("/endpoint")
        .with(oauth2Client()

.clientRegistration(this.clientRegistrationRepository.findByRegistrationId("facebo
ok"))));
```

Kotlin

```
@Autowired
lateinit var clientRegistrationRepository: ClientRegistrationRepository

// ...

mvc.get("/endpoint") {
    with(oauth2Client("my-app")

.clientRegistration(clientRegistrationRepository.findByRegistrationId("facebook"))
    )
}
```

Testing JWT Authentication

In order to make an authorized request on a resource server, you need a bearer token.

If your resource server is configured for JWTs, then this would mean that the bearer token needs to be signed and then encoded according to the JWT specification. All of this can be quite daunting, especially when this isn't the focus of your test.

Fortunately, there are a number of simple ways that you can overcome this difficulty and allow your tests to focus on authorization and not on representing bearer tokens. We'll look at two of them now:

`jwt()` `RequestPostProcessor`

The first way is via a `RequestPostProcessor`. The simplest of these would look something like this:

Java

```
mvc
    .perform(get("/endpoint").with(jwt()));
```

Kotlin

```
mvc.get("/endpoint") {
    with(jwt())
}
```

What this will do is create a mock `Jwt`, passing it correctly through any authentication APIs so that it's available for your authorization mechanisms to verify.

By default, the `JWT` that it creates has the following characteristics:

```
{
  "headers" : { "alg" : "none" },
  "claims" : {
    "sub" : "user",
    "scope" : "read"
  }
}
```

And the resulting `Jwt`, were it tested, would pass in the following way:

Java

```
assertThat(jwt.getTokenValue()).isEqualTo("token");
assertThat(jwt.getHeaders().get("alg")).isEqualTo("none");
assertThat(jwt.getSubject()).isEqualTo("sub");
```

Kotlin

```
assertThat(jwt.tokenValue).isEqualTo("token")
assertThat(jwt.headers["alg"]).isEqualTo("none")
assertThat(jwt.subject).isEqualTo("sub")
```

These values can, of course be configured.

Any headers or claims can be configured with their corresponding methods:

Java

```
mvc
    .perform(get("/endpoint")
        .with(jwt().jwt(jwt -> jwt.header("kid", "one").claim("iss",
"https://idp.example.org"))));
```

Kotlin

```
mvc.get("/endpoint") {
    with(
        jwt().jwt { jwt -> jwt.header("kid", "one").claim("iss",
"https://idp.example.org") }
    )
}
```

Java

```
mvc
    .perform(get("/endpoint")
        .with(jwt().jwt(jwt -> jwt.claims(claims -> claims.remove("scope")))));
```

Kotlin

```
mvc.get("/endpoint") {
    with(
        jwt().jwt { jwt -> jwt.claims { claims -> claims.remove("scope") } }
    )
}
```

The `scope` and `scp` claims are processed the same way here as they are in a normal bearer token request. However, this can be overridden simply by providing the list of `GrantedAuthority` instances that you need for your test:

Java

```
mvc
    .perform(get("/endpoint")
        .with(jwt().authorities(new SimpleGrantedAuthority("SCOPE_messages"))));
```

Kotlin

```
mvc.get("/endpoint") {
    with(
        jwt().authorities(SimpleGrantedAuthority("SCOPE_messages"))
    )
}
```

Or, if you have a custom `Jwt` to `Collection<GrantedAuthority>` converter, you can also use that to derive the authorities:

Java

```
mvc
    .perform(get("/endpoint")
        .with(jwt().authorities(new MyConverter())));
```

Kotlin

```
mvc.get("/endpoint") {
    with(
        jwt().authorities(MyConverter())
    )
}
```

You can also specify a complete `Jwt`, for which `Jwt.Builder` comes quite handy:

Java

```
Jwt jwt = Jwt.withTokenValue("token")
    .header("alg", "none")
    .claim("sub", "user")
    .claim("scope", "read")
    .build();

mvc
    .perform(get("/endpoint")
        .with(jwt().jwt(jwt)));
```

Kotlin

```
val jwt: Jwt = Jwt.withTokenValue("token")
    .header("alg", "none")
    .claim("sub", "user")
    .claim("scope", "read")
    .build()

mvc.get("/endpoint") {
    with(
        jwt().jwt(jwt)
    )
}
```

`authentication()` `RequestPostProcessor`

The second way is by using the `authentication()` `RequestPostProcessor`. Essentially, you can instantiate your own `JwtAuthenticationToken` and provide it in your test, like so:

Java

```
Jwt jwt = Jwt.withTokenValue("token")
    .header("alg", "none")
    .claim("sub", "user")
    .build();
Collection<GrantedAuthority> authorities =
AuthorityUtils.createAuthorityList("SCOPE_read");
JwtAuthenticationToken token = new JwtAuthenticationToken(jwt, authorities);

mvc
    .perform(get("/endpoint")
        .with(authentication(token)));
```

Kotlin

```
val jwt = Jwt.withTokenValue("token")
    .header("alg", "none")
    .claim("sub", "user")
    .build()
val authorities: Collection<GrantedAuthority> =
AuthorityUtils.createAuthorityList("SCOPE_read")
val token = JwtAuthenticationToken(jwt, authorities)

mvc.get("/endpoint") {
    with(
        authentication(token)
    )
}
```

Note that as an alternative to these, you can also mock the `JwtDecoder` bean itself with a `@MockBean` annotation.

Testing Opaque Token Authentication

Similar to JWTs, opaque tokens require an authorization server in order to verify their validity, which can make testing more difficult. To help with that, Spring Security has test support for opaque tokens.

Let's say that we've got a controller that retrieves the authentication as a `BearerTokenAuthentication`:

Java

```
@GetMapping("/endpoint")
public String foo(BearerTokenAuthentication authentication) {
    return (String) authentication.getTokenAttributes().get("sub");
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(authentication: BearerTokenAuthentication): String {
    return authentication.tokenAttributes["sub"] as String
}
```

In that case, we can tell Spring Security to include a default `BearerTokenAuthentication` using the `SecurityMockMvcRequestPostProcessors#opaqueToken` method, like so:

Java

```
mvc
    .perform(get("/endpoint").with(opaqueToken()));
```

Kotlin

```
mvc.get("/endpoint") {
    with(opaqueToken())
}
```

What this will do is configure the associated `MockHttpServletRequest` with a `BearerTokenAuthentication` that includes a simple `OAuth2AuthenticatedPrincipal`, `Map` of attributes, and `Collection` of granted authorities.

Specifically, it will include a `Map` with a key/value pair of `sub`/`user`:

Java

```
assertThat((String) token.getTokenAttributes().get("sub")).isEqualTo("user");
```

Kotlin

```
assertThat(token.tokenAttributes["sub"] as String).isEqualTo("user")
```

and a `Collection` of authorities with just one authority, `SCOPE_read`:

Java

```
assertThat(token.getAuthorities()).hasSize(1);
assertThat(token.getAuthorities()).containsExactly(new
SimpleGrantedAuthority("SCOPE_read"));
```

Kotlin

```
assertThat(token.authorities).hasSize(1)
assertThat(token.authorities).containsExactly(SimpleGrantedAuthority("SCOPE_read")
)
```

Spring Security does the necessary work to make sure that the `BearerTokenAuthentication` instance is available for your controller methods.

Configuring Authorities

In many circumstances, your method is protected by filter or method security and needs your `Authentication` to have certain granted authorities to allow the request.

In this case, you can supply what granted authorities you need using the `authorities()` method:

Java

```
mvc
    .perform(get("/endpoint")
        .with(opaqueToken()
            .authorities(new SimpleGrantedAuthority("SCOPE_message:read"))
        )
    );
```

Kotlin

```
mvc.get("/endpoint") {
    with(opaqueToken()
        .authorities(SimpleGrantedAuthority("SCOPE_message:read"))
    )
}
```

Configuring Claims

And while granted authorities are quite common across all of Spring Security, we also have attributes in the case of OAuth 2.0.

Let's say, for example, that you've got a `user_id` attribute that indicates the user's id in your system. You might access it like so in a controller:

Java

```
@GetMapping("/endpoint")
public String foo(BearerTokenAuthentication authentication) {
    String userId = (String) authentication.getTokenAttributes().get("user_id");
    // ...
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(authentication: BearerTokenAuthentication): String {
    val userId = authentication.tokenAttributes["user_id"] as String
    // ...
}
```

In that case, you'd want to specify that attribute with the `attributes()` method:

Java

```
mvc
    .perform(get("/endpoint")
        .with(opaqueToken()
                .attributes(attrs -> attrs.put("user_id", "1234"))
        )
    );
```

Kotlin

```
mvc.get("/endpoint") {
    with(opaqueToken()
        .attributes { attrs -> attrs["user_id"] = "1234" }
    )
}
```

Additional Configurations

There are additional methods, too, for further configuring the authentication; it simply depends on what data your controller expects.

One such is `principal(OAuth2AuthenticatedPrincipal)`, which you can use to configure the complete `OAuth2AuthenticatedPrincipal` instance that underlies the `BearerTokenAuthentication`

It's handy if you: 1. Have your own implementation of `OAuth2AuthenticatedPrincipal`, or 2. Want to specify a different principal name

For example, let's say that your authorization server sends the principal name in the `user_name` attribute instead of the `sub` attribute. In that case, you can configure an `OAuth2AuthenticatedPrincipal` by hand:

Java

```
Map<String, Object> attributes = Collections.singletonMap("user_name",
"foo_user");
OAuth2AuthenticatedPrincipal principal = new DefaultOAuth2AuthenticatedPrincipal(
        (String) attributes.get("user_name"),
        attributes,
        AuthorityUtils.createAuthorityList("SCOPE_message:read"));

mvc
    .perform(get("/endpoint")
        .with(opaqueToken().principal(principal))
    );
```

Kotlin

```
val attributes: Map<String, Any> = Collections.singletonMap("user_name",
"foo_user")
val principal: OAuth2AuthenticatedPrincipal = DefaultOAuth2AuthenticatedPrincipal(
    attributes["user_name"] as String?,
    attributes,
    AuthorityUtils.createAuthorityList("SCOPE_message:read")
)

mvc.get("/endpoint") {
    with(opaqueToken().principal(principal))
}
```

Note that as an alternative to using `opaqueToken()` test support, you can also mock the `OpaqueTokenIntrospector` bean itself with a `@MockBean` annotation.

19.2.3. SecurityMockMvcRequestBuilders

Spring MVC Test also provides a `RequestBuilder` interface that can be used to create the `MockHttpServletRequest` used in your test. Spring Security provides a few `RequestBuilder` implementations that can be used to make testing easier. In order to use Spring Security's `RequestBuilder` implementations ensure the following static import is used:

Java

```
import static
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestBuilde
rs.*;
```

Kotlin

```
import
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestBuilde
rs.*
```

Testing Form Based Authentication

You can easily create a request to test a form based authentication using Spring Security's testing support. For example, the following will submit a POST to "/login" with the username "user", the password "password", and a valid CSRF token:

Java

```
mvc
    .perform(formLogin())
```

Kotlin

```
mvc
    .perform(formLogin())
```

It is easy to customize the request. For example, the following will submit a POST to "/auth" with the username "admin", the password "pass", and a valid CSRF token:

Java

```
mvc
    .perform(formLogin("/auth").user("admin").password("pass"))
```

Kotlin

```
mvc
    .perform(formLogin("/auth").user("admin").password("pass"))
```

We can also customize the parameters names that the username and password are included on. For

example, this is the above request modified to include the username on the HTTP parameter "u" and the password on the HTTP parameter "p".

Java

```
mvc
    .perform(formLogin("/auth").user("u","admin").password("p","pass"))
```

Kotlin

```
mvc
    .perform(formLogin("/auth").user("u","admin").password("p","pass"))
```

Testing Logout

While fairly trivial using standard Spring MVC Test, you can use Spring Security's testing support to make testing log out easier. For example, the following will submit a POST to "/logout" with a valid CSRF token:

Java

```
mvc
    .perform(logout())
```

Kotlin

```
mvc
    .perform(logout())
```

You can also customize the URL to post to. For example, the snippet below will submit a POST to "/signout" with a valid CSRF token:

Java

```
mvc
    .perform(logout("/signout"))
```

Kotlin

```
mvc
    .perform(logout("/signout"))
```

19.2.4. SecurityMockMvcResultMatchers

At times it is desirable to make various security related assertions about a request. To accommodate this need, Spring Security Test support implements Spring MVC Test's `ResultMatcher` interface. In order to use Spring Security's `ResultMatcher` implementations ensure the following static import is used:

Java

```
import static
org.springframework.security.test.web.servlet.response.SecurityMockMvcResultMatche
rs.*;
```

Kotlin

```
import
org.springframework.security.test.web.servlet.response.SecurityMockMvcResultMatche
rs.*
```

Unauthenticated Assertion

At times it may be valuable to assert that there is no authenticated user associated with the result of a `MockMvc` invocation. For example, you might want to test submitting an invalid username and password and verify that no user is authenticated. You can easily do this with Spring Security's testing support using something like the following:

Java

```
mvc
    .perform(formLogin().password("invalid"))
    .andExpect(unauthenticated());
```

Kotlin

```
mvc
    .perform(formLogin().password("invalid"))
    .andExpect { unauthenticated() }
```

Authenticated Assertion

It is often times that we must assert that an authenticated user exists. For example, we may want to verify that we authenticated successfully. We could verify that a form based login was successful with the following snippet of code:

Java

```
mvc
    .perform(formLogin())
    .andExpect(authenticated());
```

Kotlin

```
mvc
    .perform(formLogin())
    .andExpect { authenticated() }
```

If we wanted to assert the roles of the user, we could refine our previous code as shown below:

Java

```
mvc
    .perform(formLogin().user("admin"))
    .andExpect(authenticated().withRoles("USER","ADMIN"));
```

Kotlin

```
mvc
    .perform(formLogin())
    .andExpect { authenticated().withRoles("USER","ADMIN") }
```

Alternatively, we could verify the username:

Java

```
mvc
    .perform(formLogin().user("admin"))
    .andExpect(authenticated().withUsername("admin"));
```

Kotlin

```
mvc
    .perform(formLogin().user("admin"))
    .andExpect { authenticated().withUsername("admin") }
```

We can also combine the assertions:

Java

```
mvc
    .perform(formLogin().user("admin"))
    .andExpect(authenticated().withUsername("admin").withRoles("USER", "ADMIN"));
```

Kotlin

```
mvc
    .perform(formLogin().user("admin"))
    .andExpect { authenticated().withUsername("admin").withRoles("USER", "ADMIN")
}
```

We can also make arbitrary assertions on the authentication

Java

```
mvc
    .perform(formLogin())
    .andExpect(authenticated().withAuthentication(auth ->

assertThat(auth).isInstanceOf(UsernamePasswordAuthenticationToken.class)));
```

Kotlin

```
mvc
    .perform(formLogin())
    .andExpect {
        authenticated().withAuthentication { auth ->

assertThat(auth).isInstanceOf(UsernamePasswordAuthenticationToken::class.java) }
        }
    }
```

Chapter 20. Spring Security Crypto Module

20.1. Introduction

The Spring Security Crypto module provides support for symmetric encryption, key generation, and password encoding. The code is distributed as part of the core module but has no dependencies on any other Spring Security (or Spring) code.

20.2. Encryptors

The Encryptors class provides factory methods for constructing symmetric encryptors. Using this class, you can create ByteEncryptors to encrypt data in raw byte[] form. You can also construct TextEncryptors to encrypt text strings. Encryptors are thread-safe.

20.2.1. BytesEncryptor

Use the `Encryptors.stronger` factory method to construct a BytesEncryptor:

Example 189. BytesEncryptor

Java

```
Encryptors.stronger("password", "salt");
```

Kotlin

```
Encryptors.stronger("password", "salt")
```

The "stronger" encryption method creates an encryptor using 256 bit AES encryption with Galois Counter Mode (GCM). It derives the secret key using PKCS #5's PBKDF2 (Password-Based Key Derivation Function #2). This method requires Java 6. The password used to generate the SecretKey should be kept in a secure place and not be shared. The salt is used to prevent dictionary attacks against the key in the event your encrypted data is compromised. A 16-byte random initialization vector is also applied so each encrypted message is unique.

The provided salt should be in hex-encoded String form, be random, and be at least 8 bytes in length. Such a salt may be generated using a KeyGenerator:

Example 190. Generating a key

Java

```
String salt = KeyGenerators.string().generateKey(); // generates a random 8-byte
salt that is then hex-encoded
```

Kotlin

```
val salt = KeyGenerators.string().generateKey() // generates a random 8-byte salt
that is then hex-encoded
```

Users may also use the `standard` encryption method, which is 256-bit AES in Cipher Block Chaining (CBC) Mode. This mode is not authenticated and does not provide any guarantees about the authenticity of the data. For a more secure alternative, users should prefer `Encryptors.stronger`.

20.2.2. TextEncryptor

Use the Encryptors.text factory method to construct a standard TextEncryptor:

Example 191. TextEncryptor

Java

```
Encryptors.text("password", "salt");
```

Kotlin

```
Encryptors.text("password", "salt")
```

A TextEncryptor uses a standard BytesEncryptor to encrypt text data. Encrypted results are returned as hex-encoded strings for easy storage on the filesystem or in the database.

Use the Encryptors.queryableText factory method to construct a "queryable" TextEncryptor:

Example 192. Queryable TextEncryptor

Java

```
Encryptors.queryableText("password", "salt");
```

Kotlin

```
Encryptors.queryableText("password", "salt")
```

The difference between a queryable TextEncryptor and a standard TextEncryptor has to do with initialization vector (iv) handling. The iv used in a queryable TextEncryptor#encrypt operation is shared, or constant, and is not randomly generated. This means the same text encrypted multiple times will always produce the same encryption result. This is less secure, but necessary for encrypted data that needs to be queried against. An example of queryable encrypted text would be an OAuth apiKey.

20.3. Key Generators

The KeyGenerators class provides a number of convenience factory methods for constructing different types of key generators. Using this class, you can create a BytesKeyGenerator to generate byte[] keys. You can also construct a StringKeyGenerator to generate string keys. KeyGenerators are thread-safe.

20.3.1. BytesKeyGenerator

Use the KeyGenerators.secureRandom factory methods to generate a BytesKeyGenerator backed by a SecureRandom instance:

Example 193. BytesKeyGenerator

Java

```
BytesKeyGenerator generator = KeyGenerators.secureRandom();
byte[] key = generator.generateKey();
```

Kotlin

```
val generator = KeyGenerators.secureRandom()
val key = generator.generateKey()
```

The default key length is 8 bytes. There is also a KeyGenerators.secureRandom variant that provides control over the key length:

Example 194. KeyGenerators.secureRandom

Java

```
KeyGenerators.secureRandom(16);
```

Kotlin

```
KeyGenerators.secureRandom(16)
```

Use the KeyGenerators.shared factory method to construct a BytesKeyGenerator that always

returns the same key on every invocation:

Example 195. KeyGenerators.shared

Java

```
KeyGenerators.shared(16);
```

Kotlin

```
KeyGenerators.shared(16)
```

20.3.2. StringKeyGenerator

Use the KeyGenerators.string factory method to construct a 8-byte, SecureRandom KeyGenerator that hex-encodes each key as a String:

Example 196. StringKeyGenerator

Java

```
KeyGenerators.string();
```

Kotlin

```
KeyGenerators.string()
```

20.4. Password Encoding

The password package of the spring-security-crypto module provides support for encoding passwords. `PasswordEncoder` is the central service interface and has the following signature:

```
public interface PasswordEncoder {

String encode(String rawPassword);

boolean matches(String rawPassword, String encodedPassword);
}
```

The matches method returns true if the rawPassword, once encoded, equals the encodedPassword. This method is designed to support password-based authentication schemes.

The `BCryptPasswordEncoder` implementation uses the widely supported "bcrypt" algorithm to hash the passwords. Bcrypt uses a random 16 byte salt value and is a deliberately slow algorithm, in order to hinder password crackers. The amount of work it does can be tuned using the "strength"

parameter which takes values from 4 to 31. The higher the value, the more work has to be done to calculate the hash. The default value is 10. You can change this value in your deployed system without affecting existing passwords, as the value is also stored in the encoded hash.

Example 197. BCryptPasswordEncoder

Java

```
// Create an encoder with strength 16
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(16);
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

Kotlin

```
// Create an encoder with strength 16
val encoder = BCryptPasswordEncoder(16)
val result: String = encoder.encode("myPassword")
assertTrue(encoder.matches("myPassword", result))
```

The `Pbkdf2PasswordEncoder` implementation uses PBKDF2 algorithm to hash the passwords. In order to defeat password cracking PBKDF2 is a deliberately slow algorithm and should be tuned to take about .5 seconds to verify a password on your system.

Example 198. Pbkdf2PasswordEncoder

Java

```
// Create an encoder with all the defaults
Pbkdf2PasswordEncoder encoder = new Pbkdf2PasswordEncoder();
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

Kotlin

```
// Create an encoder with all the defaults
val encoder = Pbkdf2PasswordEncoder()
val result: String = encoder.encode("myPassword")
assertTrue(encoder.matches("myPassword", result))
```

Chapter 21. Appendix

21.1. Security Database Schema

There are various database schema used by the framework and this appendix provides a single reference point to them all. You only need to provide the tables for the areas of functionality you require.

DDL statements are given for the HSQLDB database. You can use these as a guideline for defining the schema for the database you are using.

21.1.1. User Schema

The standard JDBC implementation of the `UserDetailsService` (`JdbcDaoImpl`) requires tables to load the password, account status (enabled or disabled) and a list of authorities (roles) for the user. You will need to adjust this schema to match the database dialect you are using.

```
create table users(
    username varchar_ignorecase(50) not null primary key,
    password varchar_ignorecase(50) not null,
    enabled boolean not null
);

create table authorities (
    username varchar_ignorecase(50) not null,
    authority varchar_ignorecase(50) not null,
    constraint fk_authorities_users foreign key(username) references users(username)
);
create unique index ix_auth_username on authorities (username,authority);
```

For Oracle database

```
CREATE TABLE USERS (
    USERNAME NVARCHAR2(128) PRIMARY KEY,
    PASSWORD NVARCHAR2(128) NOT NULL,
    ENABLED CHAR(1) CHECK (ENABLED IN ('Y','N') ) NOT NULL
);


CREATE TABLE AUTHORITIES (
    USERNAME NVARCHAR2(128) NOT NULL,
    AUTHORITY NVARCHAR2(128) NOT NULL
);
ALTER TABLE AUTHORITIES ADD CONSTRAINT AUTHORITIES_UNIQUE UNIQUE (USERNAME,
AUTHORITY);
ALTER TABLE AUTHORITIES ADD CONSTRAINT AUTHORITIES_FK1 FOREIGN KEY (USERNAME)
REFERENCES USERS (USERNAME) ENABLE;
```

Group Authorities

Spring Security 2.0 introduced support for group authorities in `JdbcDaoImpl`. The table structure if groups are enabled is as follows. You will need to adjust this schema to match the database dialect you are using.

```
create table groups (
    id bigint generated by default as identity(start with 0) primary key,
    group_name varchar_ignorecase(50) not null
);

create table group_authorities (
    group_id bigint not null,
    authority varchar(50) not null,
    constraint fk_group_authorities_group foreign key(group_id) references groups(id)
);

create table group_members (
    id bigint generated by default as identity(start with 0) primary key,
    username varchar(50) not null,
    group_id bigint not null,
    constraint fk_group_members_group foreign key(group_id) references groups(id)
);
```

Remember that these tables are only required if you are using the provided JDBC `UserDetailsService` implementation. If you write your own or choose to implement `AuthenticationProvider` without a `UserDetailsService`, then you have complete freedom over how you store the data, as long as the interface contract is satisfied.

21.1.2. Persistent Login (Remember-Me) Schema

This table is used to store data used by the more secure persistent token remember-me

implementation. If you are using `JdbcTokenRepositoryImpl` either directly or through the namespace, then you will need this table. Remember to adjust this schema to match the database dialect you are using.

```
create table persistent_logins (
    username varchar(64) not null,
    series varchar(64) primary key,
    token varchar(64) not null,
    last_used timestamp not null
);
```

21.1.3. ACL Schema

There are four tables used by the Spring Security ACL implementation.

1. `acl_sid` stores the security identities recognised by the ACL system. These can be unique principals or authorities which may apply to multiple principals.
2. `acl_class` defines the domain object types to which ACLs apply. The `class` column stores the Java class name of the object.
3. `acl_object_identity` stores the object identity definitions of specific domain objects.
4. `acl_entry` stores the ACL permissions which apply to a specific object identity and security identity.

It is assumed that the database will auto-generate the primary keys for each of the identities. The `JdbcMutableAclService` has to be able to retrieve these when it has created a new row in the `acl_sid` or `acl_class` tables. It has two properties which define the SQL needed to retrieve these values `classIdentityQuery` and `sidIdentityQuery`. Both of these default to `call identity()`

The ACL artifact JAR contains files for creating the ACL schema in HyperSQL (HSQLDB), PostgreSQL, MySQL/MariaDB, Microsoft SQL Server, and Oracle Database. These schemas are also demonstrated in the following sections.

HyperSQL

The default schema works with the embedded HSQLDB database that is used in unit tests within the framework.

```
create table acl_sid(
    id bigint generated by default as identity(start with 100) not null primary key,
    principal boolean not null,
    sid varchar_ignorecase(100) not null,
    constraint unique_uk_1 unique(sid,principal)
);

create table acl_class(
    id bigint generated by default as identity(start with 100) not null primary key,
    class varchar_ignorecase(100) not null,
    constraint unique_uk_2 unique(class)
);

create table acl_object_identity(
    id bigint generated by default as identity(start with 100) not null primary key,
    object_id_class bigint not null,
    object_id_identity varchar_ignorecase(36) not null,
    parent_object bigint,
    owner_sid bigint,
    entries_inheriting boolean not null,
    constraint unique_uk_3 unique(object_id_class,object_id_identity),
    constraint foreign_fk_1 foreign key(parent_object)references
acl_object_identity(id),
    constraint foreign_fk_2 foreign key(object_id_class)references acl_class(id),
    constraint foreign_fk_3 foreign key(owner_sid)references acl_sid(id)
);

create table acl_entry(
    id bigint generated by default as identity(start with 100) not null primary key,
    acl_object_identity bigint not null,
    ace_order int not null,
    sid bigint not null,
    mask integer not null,
    granting boolean not null,
    audit_success boolean not null,
    audit_failure boolean not null,
    constraint unique_uk_4 unique(acl_object_identity,ace_order),
    constraint foreign_fk_4 foreign key(acl_object_identity) references
acl_object_identity(id),
    constraint foreign_fk_5 foreign key(sid) references acl_sid(id)
);
```

PostgreSQL

```
create table acl_sid(
    id bigserial not null primary key,
    principal boolean not null,
    sid varchar(100) not null,
    constraint unique_uk_1 unique(sid,principal)
);

create table acl_class(
    id bigserial not null primary key,
    class varchar(100) not null,
    constraint unique_uk_2 unique(class)
);

create table acl_object_identity(
    id bigserial primary key,
    object_id_class bigint not null,
    object_id_identity varchar(36) not null,
    parent_object bigint,
    owner_sid bigint,
    entries_inheriting boolean not null,
    constraint unique_uk_3 unique(object_id_class,object_id_identity),
    constraint foreign_fk_1 foreign key(parent_object)references
acl_object_identity(id),
    constraint foreign_fk_2 foreign key(object_id_class)references acl_class(id),
    constraint foreign_fk_3 foreign key(owner_sid)references acl_sid(id)
);

create table acl_entry(
    id bigserial primary key,
    acl_object_identity bigint not null,
    ace_order int not null,
    sid bigint not null,
    mask integer not null,
    granting boolean not null,
    audit_success boolean not null,
    audit_failure boolean not null,
    constraint unique_uk_4 unique(acl_object_identity,ace_order),
    constraint foreign_fk_4 foreign key(acl_object_identity) references
acl_object_identity(id),
    constraint foreign_fk_5 foreign key(sid) references acl_sid(id)
);
```

You will have to set the `classIdentityQuery` and `sidIdentityQuery` properties of `JdbcMutableAclService` to the following values, respectively:

- `select currval(pg_get_serial_sequence('acl_class', 'id'))`
- `select currval(pg_get_serial_sequence('acl_sid', 'id'))`

MySQL and MariaDB

```
CREATE TABLE acl_sid (
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    principal BOOLEAN NOT NULL,
    sid VARCHAR(100) NOT NULL,
    UNIQUE KEY unique_acl_sid (sid, principal)
) ENGINE=InnoDB;

CREATE TABLE acl_class (
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    class VARCHAR(100) NOT NULL,
    UNIQUE KEY uk_acl_class (class)
) ENGINE=InnoDB;

CREATE TABLE acl_object_identity (
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    object_id_class BIGINT UNSIGNED NOT NULL,
    object_id_identity VARCHAR(36) NOT NULL,
    parent_object BIGINT UNSIGNED,
    owner_sid BIGINT UNSIGNED,
    entries_inheriting BOOLEAN NOT NULL,
    UNIQUE KEY uk_acl_object_identity (object_id_class, object_id_identity),
    CONSTRAINT fk_acl_object_identity_parent FOREIGN KEY (parent_object) REFERENCES
acl_object_identity (id),
    CONSTRAINT fk_acl_object_identity_class FOREIGN KEY (object_id_class) REFERENCES
acl_class (id),
    CONSTRAINT fk_acl_object_identity_owner FOREIGN KEY (owner_sid) REFERENCES acl_sid
(id)
) ENGINE=InnoDB;

CREATE TABLE acl_entry (
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    acl_object_identity BIGINT UNSIGNED NOT NULL,
    ace_order INTEGER NOT NULL,
    sid BIGINT UNSIGNED NOT NULL,
    mask INTEGER UNSIGNED NOT NULL,
    granting BOOLEAN NOT NULL,
    audit_success BOOLEAN NOT NULL,
    audit_failure BOOLEAN NOT NULL,
    UNIQUE KEY unique_acl_entry (acl_object_identity, ace_order),
    CONSTRAINT fk_acl_entry_object FOREIGN KEY (acl_object_identity) REFERENCES
acl_object_identity (id),
    CONSTRAINT fk_acl_entry_acl FOREIGN KEY (sid) REFERENCES acl_sid (id)
) ENGINE=InnoDB;
```

Microsoft SQL Server

```
CREATE TABLE acl_sid (
    id BIGINT NOT NULL IDENTITY PRIMARY KEY,
    principal BIT NOT NULL,
    sid VARCHAR(100) NOT NULL,
    CONSTRAINT unique_acl_sid UNIQUE (sid, principal)
);

CREATE TABLE acl_class (
    id BIGINT NOT NULL IDENTITY PRIMARY KEY,
    class VARCHAR(100) NOT NULL,
    CONSTRAINT uk_acl_class UNIQUE (class)
);

CREATE TABLE acl_object_identity (
    id BIGINT NOT NULL IDENTITY PRIMARY KEY,
    object_id_class BIGINT NOT NULL,
    object_id_identity VARCHAR(36) NOT NULL,
    parent_object BIGINT,
    owner_sid BIGINT,
    entries_inheriting BIT NOT NULL,
    CONSTRAINT uk_acl_object_identity UNIQUE (object_id_class, object_id_identity),
    CONSTRAINT fk_acl_object_identity_parent FOREIGN KEY (parent_object) REFERENCES
acl_object_identity (id),
    CONSTRAINT fk_acl_object_identity_class FOREIGN KEY (object_id_class) REFERENCES
acl_class (id),
    CONSTRAINT fk_acl_object_identity_owner FOREIGN KEY (owner_sid) REFERENCES acl_sid
(id)
);

CREATE TABLE acl_entry (
    id BIGINT NOT NULL IDENTITY PRIMARY KEY,
    acl_object_identity BIGINT NOT NULL,
    ace_order INTEGER NOT NULL,
    sid BIGINT NOT NULL,
    mask INTEGER NOT NULL,
    granting BIT NOT NULL,
    audit_success BIT NOT NULL,
    audit_failure BIT NOT NULL,
    CONSTRAINT unique_acl_entry UNIQUE (acl_object_identity, ace_order),
    CONSTRAINT fk_acl_entry_object FOREIGN KEY (acl_object_identity) REFERENCES
acl_object_identity (id),
    CONSTRAINT fk_acl_entry_acl FOREIGN KEY (sid) REFERENCES acl_sid (id)
);
```

Oracle Database

```
CREATE TABLE ACL_SID (
    ID NUMBER(18) PRIMARY KEY,
    PRINCIPAL NUMBER(1) NOT NULL CHECK (PRINCIPAL IN (0, 1 )),
```

```
    SID NVARCHAR2(128) NOT NULL,
    CONSTRAINT ACL_SID_UNIQUE UNIQUE (SID, PRINCIPAL)
);
CREATE SEQUENCE ACL_SID_SQ START WITH 1 INCREMENT BY 1 NOMAXVALUE;
CREATE OR REPLACE TRIGGER ACL_SID_SQ_TR BEFORE INSERT ON ACL_SID FOR EACH ROW
BEGIN
    SELECT ACL_SID_SQ.NEXTVAL INTO :NEW.ID FROM DUAL;
END;


CREATE TABLE ACL_CLASS (
    ID NUMBER(18) PRIMARY KEY,
    CLASS NVARCHAR2(128) NOT NULL,
    CONSTRAINT ACL_CLASS_UNIQUE UNIQUE (CLASS)
);
CREATE SEQUENCE ACL_CLASS_SQ START WITH 1 INCREMENT BY 1 NOMAXVALUE;
CREATE OR REPLACE TRIGGER ACL_CLASS_ID_TR BEFORE INSERT ON ACL_CLASS FOR EACH ROW
BEGIN
    SELECT ACL_CLASS_SQ.NEXTVAL INTO :NEW.ID FROM DUAL;
END;


CREATE TABLE ACL_OBJECT_IDENTITY(
    ID NUMBER(18) PRIMARY KEY,
    OBJECT_ID_CLASS NUMBER(18) NOT NULL,
    OBJECT_ID_IDENTITY NVARCHAR2(64) NOT NULL,
    PARENT_OBJECT NUMBER(18),
    OWNER_SID NUMBER(18),
    ENTRIES_INHERITING NUMBER(1) NOT NULL CHECK (ENTRIES_INHERITING IN (0, 1)),
    CONSTRAINT ACL_OBJECT_IDENTITY_UNIQUE UNIQUE (OBJECT_ID_CLASS,
OBJECT_ID_IDENTITY),
    CONSTRAINT ACL_OBJECT_IDENTITY_PARENT_FK FOREIGN KEY (PARENT_OBJECT) REFERENCES
ACL_OBJECT_IDENTITY(ID),
    CONSTRAINT ACL_OBJECT_IDENTITY_CLASS_FK FOREIGN KEY (OBJECT_ID_CLASS) REFERENCES
ACL_CLASS(ID),
    CONSTRAINT ACL_OBJECT_IDENTITY_OWNER_FK FOREIGN KEY (OWNER_SID) REFERENCES
ACL_SID(ID)
);
CREATE SEQUENCE ACL_OBJECT_IDENTITY_SQ START WITH 1 INCREMENT BY 1 NOMAXVALUE;
CREATE OR REPLACE TRIGGER ACL_OBJECT_IDENTITY_ID_TR BEFORE INSERT ON
ACL_OBJECT_IDENTITY FOR EACH ROW
BEGIN
    SELECT ACL_OBJECT_IDENTITY_SQ.NEXTVAL INTO :NEW.ID FROM DUAL;
END;


CREATE TABLE ACL_ENTRY (
    ID NUMBER(18) NOT NULL PRIMARY KEY,
    ACL_OBJECT_IDENTITY NUMBER(18) NOT NULL,
    ACE_ORDER INTEGER NOT NULL,
    SID NUMBER(18) NOT NULL,
```

```
    MASK INTEGER NOT NULL,
    GRANTING NUMBER(1) NOT NULL CHECK (GRANTING IN (0, 1)),
    AUDIT_SUCCESS NUMBER(1) NOT NULL CHECK (AUDIT_SUCCESS IN (0, 1)),
    AUDIT_FAILURE NUMBER(1) NOT NULL CHECK (AUDIT_FAILURE IN (0, 1)),
    CONSTRAINT ACL_ENTRY_UNIQUE UNIQUE (ACL_OBJECT_IDENTITY, ACE_ORDER),
    CONSTRAINT ACL_ENTRY_OBJECT_FK FOREIGN KEY (ACL_OBJECT_IDENTITY) REFERENCES
ACL_OBJECT_IDENTITY (ID),
    CONSTRAINT ACL_ENTRY_ACL_FK FOREIGN KEY (SID) REFERENCES ACL_SID(ID)
);
CREATE SEQUENCE ACL_ENTRY_SQ START WITH 1 INCREMENT BY 1 NOMAXVALUE;
CREATE OR REPLACE TRIGGER ACL_ENTRY_ID_TRIGGER BEFORE INSERT ON ACL_ENTRY FOR EACH ROW
BEGIN
    SELECT ACL_ENTRY_SQ.NEXTVAL INTO :NEW.ID FROM DUAL;
END;
```

21.1.4. OAuth 2.0 Client Schema

The JDBC implementation of OAuth2AuthorizedClientService (`JdbcOAuth2AuthorizedClientService`) requires a table for persisting `OAuth2AuthorizedClient`(s). You will need to adjust this schema to match the database dialect you are using.

```
CREATE TABLE oauth2_authorized_client (
  client_registration_id varchar(100) NOT NULL,
  principal_name varchar(200) NOT NULL,
  access_token_type varchar(100) NOT NULL,
  access_token_value blob NOT NULL,
  access_token_issued_at timestamp NOT NULL,
  access_token_expires_at timestamp NOT NULL,
  access_token_scopes varchar(1000) DEFAULT NULL,
  refresh_token_value blob DEFAULT NULL,
  refresh_token_issued_at timestamp DEFAULT NULL,
  created_at timestamp DEFAULT CURRENT_TIMESTAMP NOT NULL,
  PRIMARY KEY (client_registration_id, principal_name)
);
```

21.2. The Security Namespace

This appendix provides a reference to the elements available in the security namespace and information on the underlying beans they create (a knowledge of the individual classes and how they work together is assumed - you can find more information in the project Javadoc and elsewhere in this document). If you haven't used the namespace before, please read the introductory chapter on namespace configuration, as this is intended as a supplement to the information there. Using a good quality XML editor while editing a configuration based on the schema is recommended as this will provide contextual information on which elements and attributes are available as well as comments explaining their purpose. The namespace is written in RELAX NG Compact format and later converted into an XSD schema. If you are familiar with this format, you may wish to examine the schema file directly.

21.2.1. Web Application Security

<debug>

Enables Spring Security debugging infrastructure. This will provide human-readable (multi-line) debugging information to monitor requests coming into the security filters. This may include sensitive information, such as request parameters or headers, and should only be used in a development environment.

<http>

If you use an `<http>` element within your application, a `FilterChainProxy` bean named "springSecurityFilterChain" is created and the configuration within the element is used to build a filter chain within `FilterChainProxy`. As of Spring Security 3.1, additional `http` elements can be used to add extra filter chains [13]. Some core filters are always created in a filter chain and others will be added to the stack depending on the attributes and child elements which are present. The positions of the standard filters are fixed (see the filter order table in the namespace introduction), removing a common source of errors with previous versions of the framework when users had to configure the filter chain explicitly in the `FilterChainProxy` bean. You can, of course, still do this if you need full control of the configuration.

All filters which require a reference to the `AuthenticationManager` will be automatically injected with the internal instance created by the namespace configuration.

Each `<http>` namespace block always creates an `SecurityContextPersistenceFilter`, an `ExceptionTranslationFilter` and a `FilterSecurityInterceptor`. These are fixed and cannot be replaced with alternatives.

<http> Attributes

The attributes on the `<http>` element control some of the properties on the core filters.

- **access-decision-manager-ref** Optional attribute specifying the ID of the `AccessDecisionManager` implementation which should be used for authorizing HTTP requests. By default an `AffirmativeBased` implementation is used for with a `RoleVoter` and an `AuthenticatedVoter`.
- **authentication-manager-ref** A reference to the `AuthenticationManager` used for the `FilterChain` created by this http element.
- **auto-config** Automatically registers a login form, BASIC authentication, logout services. If set to "true", all of these capabilities are added (although you can still customize the configuration of each by providing the respective element). If unspecified, defaults to "false". Use of this attribute is not recommended. Use explicit configuration elements instead to avoid confusion.
- **create-session** Controls the eagerness with which an HTTP session is created by Spring Security classes. Options include:
  - `always` - Spring Security will proactively create a session if one does not exist.
  - `ifRequired` - Spring Security will only create a session only if one is required (default value).
  - `never` - Spring Security will never create a session, but will make use of one if the application

does.

  - `stateless` - Spring Security will not create a session and ignore the session for obtaining a Spring `Authentication`.

- **disable-url-rewriting** Prevents session IDs from being appended to URLs in the application. Clients must use cookies if this attribute is set to `true`. The default is `true`.

- **entry-point-ref** Normally the `AuthenticationEntryPoint` used will be set depending on which authentication mechanisms have been configured. This attribute allows this behaviour to be overridden by defining a customized `AuthenticationEntryPoint` bean which will start the authentication process.

- **jaas-api-provision** If available, runs the request as the `Subject` acquired from the `JaasAuthenticationToken` which is implemented by adding a `JaasApiIntegrationFilter` bean to the stack. Defaults to `false`.

- **name** A bean identifier, used for referring to the bean elsewhere in the context.

- **once-per-request** Corresponds to the `observeOncePerRequest` property of `FilterSecurityInterceptor`. Defaults to `true`.

- **pattern** Defining a pattern for the http element controls the requests which will be filtered through the list of filters which it defines. The interpretation is dependent on the configured request-matcher. If no pattern is defined, all requests will be matched, so the most specific patterns should be declared first.

- **realm** Sets the realm name used for basic authentication (if enabled). Corresponds to the `realmName` property on `BasicAuthenticationEntryPoint`.

- **request-matcher** Defines the `RequestMatcher` strategy used in the `FilterChainProxy` and the beans created by the `intercept-url` to match incoming requests. Options are currently `mvc`, `ant`, `regex` and `ciRegex`, for Spring MVC, ant, regular-expression and case-insensitive regular-expression respectively. A separate instance is created for each intercept-url element using its pattern, method and servlet-path attributes. Ant paths are matched using an `AntPathRequestMatcher`, regular expressions are matched using a `RegexRequestMatcher` and for Spring MVC path matching the `MvcRequestMatcher` is used. See the Javadoc for these classes for more details on exactly how the matching is performed. Ant paths are the default strategy.

- **request-matcher-ref** A reference to a bean that implements `RequestMatcher` that will determine if this `FilterChain` should be used. This is a more powerful alternative to pattern.

- **security** A request pattern can be mapped to an empty filter chain, by setting this attribute to `none`. No security will be applied and none of Spring Security's features will be available.

- **security-context-repository-ref** Allows injection of a custom `SecurityContextRepository` into the `SecurityContextPersistenceFilter`.

- **servlet-api-provision** Provides versions of `HttpServletRequest` security methods such as `isUserInRole()` and `getPrincipal()` which are implemented by adding a `SecurityContextHolderAwareRequestFilter` bean to the stack. Defaults to `true`.

- **use-expressions** Enables EL-expressions in the `access` attribute, as described in the chapter on expression-based access-control. The default value is true.

Child Elements of <http>

- access-denied-handler
- anonymous
- cors
- csrf
- custom-filter
- expression-handler
- form-login
- headers
- http-basic
- intercept-url
- jee
- logout
- oauth2-client
- oauth2-login
- oauth2-resource-server
- openid-login
- password-management
- port-mappings
- remember-me
- request-cache
- session-management
- x509

<access-denied-handler>

This element allows you to set the `errorPage` property for the default `AccessDeniedHandler` used by the `ExceptionTranslationFilter`, using the error-page attribute, or to supply your own implementation using theref attribute. This is discussed in more detail in the section on the ExceptionTranslationFilter.

Parent Elements of <access-denied-handler>

- http

<access-denied-handler> Attributes

- **error-page** The access denied page that an authenticated user will be redirected to if they request a page which they don't have the authority to access.
- **ref** Defines a reference to a Spring bean of type `AccessDeniedHandler`.

<cors>

This element allows for configuring a `CorsFilter`. If no `CorsFilter` or `CorsConfigurationSource` is specified and Spring MVC is on the classpath, a `HandlerMappingIntrospector` is used as the `CorsConfigurationSource`.

<cors> Attributes

The attributes on the `<cors>` element control the headers element.

- **ref** Optional attribute that specifies the bean name of a `CorsFilter`.
- **cors-configuration-source-ref** Optional attribute that specifies the bean name of a `CorsConfigurationSource` to be injected into a `CorsFilter` created by the XML namespace.

Parent Elements of <cors>

- http

<headers>

This element allows for configuring additional (security) headers to be send with the response. It enables easy configuration for several headers and also allows for setting custom headers through the header element. Additional information, can be found in the Security Headers section of the reference.

- `Cache-Control`, `Pragma`, and `Expires` - Can be set using the cache-control element. This ensures that the browser does not cache your secured pages.
- `Strict-Transport-Security` - Can be set using the hsts element. This ensures that the browser automatically requests HTTPS for future requests.
- `X-Frame-Options` - Can be set using the frame-options element. The X-Frame-Options header can be used to prevent clickjacking attacks.
- `X-XSS-Protection` - Can be set using the xss-protection element. The X-XSS-Protection header can be used by browser to do basic control.
- `X-Content-Type-Options` - Can be set using the content-type-options element. The X-Content-Type-Options header prevents Internet Explorer from MIME-sniffing a response away from the declared content-type. This also applies to Google Chrome, when downloading extensions.
- `Public-Key-Pinning` or `Public-Key-Pinning-Report-Only` - Can be set using the hpkp element. This allows HTTPS websites to resist impersonation by attackers using mis-issued or otherwise fraudulent certificates.
- `Content-Security-Policy` or `Content-Security-Policy-Report-Only` - Can be set using the content-

security-policy element. Content Security Policy (CSP) is a mechanism that web applications can leverage to mitigate content injection vulnerabilities, such as cross-site scripting (XSS).

- `Referrer-Policy` - Can be set using the referrer-policy element, Referrer-Policy is a mechanism that web applications can leverage to manage the referrer field, which contains the last page the user was on.
- `Feature-Policy` - Can be set using the feature-policy element, Feature-Policy is a mechanism that allows web developers to selectively enable, disable, and modify the behavior of certain APIs and web features in the browser.

<headers> Attributes

The attributes on the `<headers>` element control the headers element.

- **defaults-disabled** Optional attribute that specifies to disable the default Spring Security's HTTP response headers. The default is false (the default headers are included).
- **disabled** Optional attribute that specifies to disable Spring Security's HTTP response headers. The default is false (the headers are enabled).

Parent Elements of <headers>

- http

Child Elements of <headers>

- cache-control
- content-security-policy
- content-type-options
- feature-policy
- frame-options
- header
- hpkp
- hsts
- permission-policy
- referrer-policy
- xss-protection

<cache-control>

Adds `Cache-Control`, `Pragma`, and `Expires` headers to ensure that the browser does not cache your secured pages.

<cache-control> Attributes

- **disabled** Specifies if Cache Control should be disabled. Default false.

Parent Elements of <cache-control>

- headers

<hsts>

When enabled adds the Strict-Transport-Security header to the response for any secure request. This allows the server to instruct browsers to automatically use HTTPS for future requests.

<hsts> Attributes

- **disabled** Specifies if Strict-Transport-Security should be disabled. Default false.
- **include-sub-domains** Specifies if subdomains should be included. Default true.
- **max-age-seconds** Specifies the maximum amount of time the host should be considered a Known HSTS Host. Default one year.
- **request-matcher-ref** The RequestMatcher instance to be used to determine if the header should be set. Default is if HttpServletRequest.isSecure() is true.
- **preload** Specifies if preload should be included. Default false.

Parent Elements of <hsts>

- headers

<hpkp>

When enabled adds the Public Key Pinning Extension for HTTP header to the response for any secure request. This allows HTTPS websites to resist impersonation by attackers using mis-issued or otherwise fraudulent certificates.

<hpkp> Attributes

- **disabled** Specifies if HTTP Public Key Pinning (HPKP) should be disabled. Default true.
- **include-sub-domains** Specifies if subdomains should be included. Default false.
- **max-age-seconds** Sets the value for the max-age directive of the Public-Key-Pins header. Default 60 days.
- **report-only** Specifies if the browser should only report pin validation failures. Default true.
- **report-uri** Specifies the URI to which the browser should report pin validation failures.

Parent Elements of <hpkp>

- headers

<pins>

The list of pins

Child Elements of <pins>

- pin

<pin>

A pin is specified using the base64-encoded SPKI fingerprint as value and the cryptographic hash algorithm as attribute

<pin> Attributes

- **algorithm** The cryptographic hash algorithm. Default is SHA256.

Parent Elements of <pin>

- pins

<content-security-policy>

When enabled adds the Content Security Policy (CSP) header to the response. CSP is a mechanism that web applications can leverage to mitigate content injection vulnerabilities, such as cross-site scripting (XSS).

<content-security-policy> Attributes

- **policy-directives** The security policy directive(s) for the Content-Security-Policy header or if report-only is set to true, then the Content-Security-Policy-Report-Only header is used.
- **report-only** Set to true, to enable the Content-Security-Policy-Report-Only header for reporting policy violations only. Defaults to false.

Parent Elements of <content-security-policy>

- headers

<referrer-policy>

When enabled adds the Referrer Policy header to the response.

<referrer-policy> Attributes

- **policy** The policy for the Referrer-Policy header. Default "no-referrer".

Parent Elements of <referrer-policy>

- headers

<feature-policy>

When enabled adds the Feature Policy header to the response.

<feature-policy> Attributes

- **policy-directives** The security policy directive(s) for the Feature-Policy header.

Parent Elements of <feature-policy>

- headers

<frame-options>

When enabled adds the X-Frame-Options header to the response, this allows newer browsers to do some security checks and prevent clickjacking attacks.

<frame-options> Attributes

- **disabled** If disabled, the X-Frame-Options header will not be included. Default false.
- **policy**
  - `DENY` The page cannot be displayed in a frame, regardless of the site attempting to do so. This is the default when frame-options-policy is specified.
  - `SAMEORIGIN` The page can only be displayed in a frame on the same origin as the page itself

  In other words, if you specify DENY, not only will attempts to load the page in a frame fail when loaded from other sites, attempts to do so will fail when loaded from the same site. On the other hand, if you specify SAMEORIGIN, you can still use the page in a frame as long as the site including it in a frame it is the same as the one serving the page.

Parent Elements of <frame-options>

- headers

<permissions-policy>

Adds the Permissions-Policy header to the response.

<permissions-policy> Attributes

- **policy** The policy value to write for the `Permissions-Policy` header

Parent Elements of <permissions-policy>

- headers

<xss-protection>

Adds the X-XSS-Protection header to the response to assist in protecting against reflected / Type-1 Cross-Site Scripting (XSS) attacks. This is in no-way a full protection to XSS attacks!

<xss-protection> Attributes

- **xss-protection-disabled** Do not include the header for reflected / Type-1 Cross-Site Scripting

(XSS) protection.

- **xss-protection-enabled** Explicitly enable or disable reflected / Type-1 Cross-Site Scripting (XSS) protection.
- **xss-protection-block** When true and xss-protection-enabled is true, adds mode=block to the header. This indicates to the browser that the page should not be loaded at all. When false and xss-protection-enabled is true, the page will still be rendered when an reflected attack is detected but the response will be modified to protect against the attack. Note that there are sometimes ways of bypassing this mode which can often times make blocking the page more desirable.

Parent Elements of <xss-protection>

- headers

<content-type-options>

Add the X-Content-Type-Options header with the value of nosniff to the response. This disables MIME-sniffing for IE8+ and Chrome extensions.

<content-type-options> Attributes

- **disabled** Specifies if Content Type Options should be disabled. Default false.

Parent Elements of <content-type-options>

- headers

<header>

Add additional headers to the response, both the name and value need to be specified.

<header-attributes> Attributes

- **header-name** The `name` of the header.
- **value** The `value` of the header to add.
- **ref** Reference to a custom implementation of the `HeaderWriter` interface.

Parent Elements of <header>

- headers

<anonymous>

Adds an `AnonymousAuthenticationFilter` to the stack and an `AnonymousAuthenticationProvider`. Required if you are using the `IS_AUTHENTICATED_ANONYMOUSLY` attribute.

Parent Elements of <anonymous>

- http

<anonymous> Attributes

- **enabled** With the default namespace setup, the anonymous "authentication" facility is automatically enabled. You can disable it using this property.
- **granted-authority** The granted authority that should be assigned to the anonymous request. Commonly this is used to assign the anonymous request particular roles, which can subsequently be used in authorization decisions. If unset, defaults to `ROLE_ANONYMOUS`.
- **key** The key shared between the provider and filter. This generally does not need to be set. If unset, it will default to a secure randomly generated value. This means setting this value can improve startup time when using the anonymous functionality since secure random values can take a while to be generated.
- **username** The username that should be assigned to the anonymous request. This allows the principal to be identified, which may be important for logging and auditing. if unset, defaults to `anonymousUser`.

<csrf>

This element will add Cross Site Request Forger (CSRF) protection to the application. It also updates the default RequestCache to only replay "GET" requests upon successful authentication. Additional information can be found in the Cross Site Request Forgery (CSRF) section of the reference.

Parent Elements of <csrf>

- http

<csrf> Attributes

- **disabled** Optional attribute that specifies to disable Spring Security's CSRF protection. The default is false (CSRF protection is enabled). It is highly recommended to leave CSRF protection enabled.
- **token-repository-ref** The CsrfTokenRepository to use. The default is `HttpSessionCsrfTokenRepository`.
- **request-matcher-ref** The RequestMatcher instance to be used to determine if CSRF should be applied. Default is any HTTP method except "GET", "TRACE", "HEAD", "OPTIONS".

<custom-filter>

This element is used to add a filter to the filter chain. It doesn't create any additional beans but is used to select a bean of type `javax.servlet.Filter` which is already defined in the application context and add that at a particular position in the filter chain maintained by Spring Security. Full details can be found in the namespace chapter.

Parent Elements of <custom-filter>

- http

<custom-filter> Attributes

- **after** The filter immediately after which the custom-filter should be placed in the chain. This feature will only be needed by advanced users who wish to mix their own filters into the security filter chain and have some knowledge of the standard Spring Security filters. The filter names map to specific Spring Security implementation filters.
- **before** The filter immediately before which the custom-filter should be placed in the chain
- **position** The explicit position at which the custom-filter should be placed in the chain. Use if you are replacing a standard filter.
- **ref** Defines a reference to a Spring bean that implements `Filter`.

<expression-handler>

Defines the `SecurityExpressionHandler` instance which will be used if expression-based access-control is enabled. A default implementation (with no ACL support) will be used if not supplied.

Parent Elements of <expression-handler>

- global-method-security
- http
- method-security
- websocket-message-broker

<expression-handler> Attributes

- **ref** Defines a reference to a Spring bean that implements `SecurityExpressionHandler`.

<form-login>

Used to add an `UsernamePasswordAuthenticationFilter` to the filter stack and an `LoginUrlAuthenticationEntryPoint` to the application context to provide authentication on demand. This will always take precedence over other namespace-created entry points. If no attributes are supplied, a login page will be generated automatically at the URL "/login" [14] The behaviour can be customized using the `<form-login>` Attributes.

Parent Elements of <form-login>

- http

<form-login> Attributes

- **always-use-default-target** If set to `true`, the user will always start at the value given by default-target-url, regardless of how they arrived at the login page. Maps to the `alwaysUseDefaultTargetUrl` property of `UsernamePasswordAuthenticationFilter`. Default value is

`false`.

- **authentication-details-source-ref** Reference to an `AuthenticationDetailsSource` which will be used by the authentication filter
- **authentication-failure-handler-ref** Can be used as an alternative to authentication-failure-url, giving you full control over the navigation flow after an authentication failure. The value should be the name of an `AuthenticationFailureHandler` bean in the application context.
- **authentication-failure-url** Maps to the `authenticationFailureUrl` property of `UsernamePasswordAuthenticationFilter`. Defines the URL the browser will be redirected to on login failure. Defaults to `/login?error`, which will be automatically handled by the automatic login page generator, re-rendering the login page with an error message.
- **authentication-success-handler-ref** This can be used as an alternative to default-target-url and always-use-default-target, giving you full control over the navigation flow after a successful authentication. The value should be the name of an `AuthenticationSuccessHandler` bean in the application context. By default, an implementation of `SavedRequestAwareAuthenticationSuccessHandler` is used and injected with the default-target-url .
- **default-target-url** Maps to the `defaultTargetUrl` property of `UsernamePasswordAuthenticationFilter`. If not set, the default value is "/" (the application root). A user will be taken to this URL after logging in, provided they were not asked to login while attempting to access a secured resource, when they will be taken to the originally requested URL.
- **login-page** The URL that should be used to render the login page. Maps to the `loginFormUrl` property of the `LoginUrlAuthenticationEntryPoint`. Defaults to "/login".
- **login-processing-url** Maps to the `filterProcessesUrl` property of `UsernamePasswordAuthenticationFilter`. The default value is "/login".
- **password-parameter** The name of the request parameter which contains the password. Defaults to "password".
- **username-parameter** The name of the request parameter which contains the username. Defaults to "username".
- **authentication-success-forward-url** Maps a `ForwardAuthenticationSuccessHandler` to `authenticationSuccessHandler` property of `UsernamePasswordAuthenticationFilter`.
- **authentication-failure-forward-url** Maps a `ForwardAuthenticationFailureHandler` to `authenticationFailureHandler` property of `UsernamePasswordAuthenticationFilter`.

<oauth2-login>

The OAuth 2.0 Login feature configures authentication support using an OAuth 2.0 and/or OpenID Connect 1.0 Provider.

Parent Elements of <oauth2-login>

- http

<oauth2-login> Attributes

- **client-registration-repository-ref** Reference to the `ClientRegistrationRepository`.
- **authorized-client-repository-ref** Reference to the `OAuth2AuthorizedClientRepository`.
- **authorized-client-service-ref** Reference to the `OAuth2AuthorizedClientService`.
- **authorization-request-repository-ref** Reference to the `AuthorizationRequestRepository`.
- **authorization-request-resolver-ref** Reference to the `OAuth2AuthorizationRequestResolver`.
- **access-token-response-client-ref** Reference to the `OAuth2AccessTokenResponseClient`.
- **user-authorities-mapper-ref** Reference to the `GrantedAuthoritiesMapper`.
- **user-service-ref** Reference to the `OAuth2UserService`.
- **oidc-user-service-ref** Reference to the OpenID Connect `OAuth2UserService`.
- **login-processing-url** The URI where the filter processes authentication requests.
- **login-page** The URI to send users to login.
- **authentication-success-handler-ref** Reference to the `AuthenticationSuccessHandler`.
- **authentication-failure-handler-ref** Reference to the `AuthenticationFailureHandler`.
- **jwt-decoder-factory-ref** Reference to the `JwtDecoderFactory` used by `OidcAuthorizationCodeAuthenticationProvider`.

<oauth2-client>

Configures OAuth 2.0 Client support.

Parent Elements of <oauth2-client>

- http

<oauth2-client> Attributes

- **client-registration-repository-ref** Reference to the `ClientRegistrationRepository`.
- **authorized-client-repository-ref** Reference to the `OAuth2AuthorizedClientRepository`.
- **authorized-client-service-ref** Reference to the `OAuth2AuthorizedClientService`.

Child Elements of <oauth2-client>

- authorization-code-grant

<authorization-code-grant>

Configures OAuth 2.0 Authorization Code Grant.

Parent Elements of <authorization-code-grant>

- oauth2-client

<authorization-code-grant> Attributes

- **authorization-request-repository-ref** Reference to the `AuthorizationRequestRepository`.
- **authorization-request-resolver-ref** Reference to the `OAuth2AuthorizationRequestResolver`.
- **access-token-response-client-ref** Reference to the `OAuth2AccessTokenResponseClient`.

<client-registrations>

A container element for client(s) registered (ClientRegistration) with an OAuth 2.0 or OpenID Connect 1.0 Provider.

Child Elements of <client-registrations>

- client-registration
- provider

<client-registration>

Represents a client registered with an OAuth 2.0 or OpenID Connect 1.0 Provider.

Parent Elements of <client-registration>

- client-registrations

<client-registration> Attributes

- **registration-id** The ID that uniquely identifies the `ClientRegistration`.
- **client-id** The client identifier.
- **client-secret** The client secret.
- **client-authentication-method** The method used to authenticate the Client with the Provider. The supported values are **client_secret_basic**, **client_secret_post**, **private_key_jwt**, **client_secret_jwt** and **none** (public clients).
- **authorization-grant-type** The OAuth 2.0 Authorization Framework defines four Authorization Grant types. The supported values are `authorization_code`, `client_credentials`, `password`, as well as, extension grant type `urn:ietf:params:oauth:grant-type:jwt-bearer`.
- **redirect-uri** The client's registered redirect URI that the *Authorization Server* redirects the end-user's user-agent to after the end-user has authenticated and authorized access to the client.

- **scope** The scope(s) requested by the client during the Authorization Request flow, such as openid, email, or profile.
- **client-name** A descriptive name used for the client. The name may be used in certain scenarios, such as when displaying the name of the client in the auto-generated login page.
- **provider-id** A reference to the associated provider. May reference a `<provider>` element or use one of the common providers (google, github, facebook, okta).

<provider>

The configuration information for an OAuth 2.0 or OpenID Connect 1.0 Provider.

Parent Elements of <provider>

- client-registrations

<provider> Attributes

- **provider-id** The ID that uniquely identifies the provider.
- **authorization-uri** The Authorization Endpoint URI for the Authorization Server.
- **token-uri** The Token Endpoint URI for the Authorization Server.
- **user-info-uri** The UserInfo Endpoint URI used to access the claims/attributes of the authenticated end-user.
- **user-info-authentication-method** The authentication method used when sending the access token to the UserInfo Endpoint. The supported values are **header**, **form** and **query**.
- **user-info-user-name-attribute** The name of the attribute returned in the UserInfo Response that references the Name or Identifier of the end-user.
- **jwk-set-uri** The URI used to retrieve the JSON Web Key (JWK) Set from the Authorization Server, which contains the cryptographic key(s) used to verify the JSON Web Signature (JWS) of the ID Token and optionally the UserInfo Response.
- **issuer-uri** The URI used to initially configure a `ClientRegistration` using discovery of an OpenID Connect Provider's Configuration endpoint or an Authorization Server's Metadata endpoint.

<oauth2-resource-server>

Adds a `BearerTokenAuthenticationFilter`, `BearerTokenAuthenticationEntryPoint`, and `BearerTokenAccessDeniedHandler` to the configuration. In addition, either `<jwt>` or `<opaque-token>` must be specified.

Parents Elements of <oauth2-resource-server>

- http

Child Elements of <oauth2-resource-server>

- jwt
- opaque-token

<oauth2-resource-server> Attributes

- **authentication-manager-resolver-ref** Reference to an `AuthenticationManagerResolver` which will resolve the `AuthenticationManager` at request time
- **bearer-token-resolver-ref** Reference to a `BearerTokenResolver` which will retrieve the bearer token from the request
- **entry-point-ref** Reference to a `AuthenticationEntryPoint` which will handle unauthorized requests

<jwt>

Represents an OAuth 2.0 Resource Server that will authorize JWTs

Parent Elements of <jwt>

- oauth2-resource-server

<jwt> Attributes

- **jwt-authentication-converter-ref** Reference to a `Converter<Jwt, AbstractAuthenticationToken>`
- **jwt-decoder-ref** Reference to a `JwtDecoder`. This is a larger component that overrides `jwk-set-uri`
- **jwk-set-uri** The JWK Set Uri used to load signing verification keys from an OAuth 2.0 Authorization Server

<opaque-token>

Represents an OAuth 2.0 Resource Server that will authorize opaque tokens

Parent Elements of <opaque-token>

- oauth2-resource-server

<opaque-token> Attributes

- **introspector-ref** Reference to an `OpaqueTokenIntrospector`. This is a larger component that overrides `introspection-uri`, `client-id`, and `client-secret`.
- **introspection-uri** The Introspection Uri used to introspect the details of an opaque token. Should be accompanied with a `client-id` and `client-secret`.
- **client-id** The Client Id to use for client authentication against the provided `introspection-uri`.
- **client-secret** The Client Secret to use for client authentication against the provided

`introspection-uri`.

<http-basic>

Adds a `BasicAuthenticationFilter` and `BasicAuthenticationEntryPoint` to the configuration. The latter will only be used as the configuration entry point if form-based login is not enabled.

Parent Elements of <http-basic>

- http

<http-basic> Attributes

- **authentication-details-source-ref** Reference to an `AuthenticationDetailsSource` which will be used by the authentication filter
- **entry-point-ref** Sets the `AuthenticationEntryPoint` which is used by the `BasicAuthenticationFilter`.

<http-firewall> Element

This is a top-level element which can be used to inject a custom implementation of `HttpFirewall` into the `FilterChainProxy` created by the namespace. The default implementation should be suitable for most applications.

<http-firewall> Attributes

- **ref** Defines a reference to a Spring bean that implements `HttpFirewall`.

<intercept-url>

This element is used to define the set of URL patterns that the application is interested in and to configure how they should be handled. It is used to construct the `FilterInvocationSecurityMetadataSource` used by the `FilterSecurityInterceptor`. It is also responsible for configuring a `ChannelProcessingFilter` if particular URLs need to be accessed by HTTPS, for example. When matching the specified patterns against an incoming request, the matching is done in the order in which the elements are declared. So the most specific patterns should come first and the most general should come last.

Parent Elements of <intercept-url>

- filter-security-metadata-source
- http

<intercept-url> Attributes

- **access** Lists the access attributes which will be stored in the `FilterInvocationSecurityMetadataSource` for the defined URL pattern/method combination. This should be a comma-separated list of the security configuration attributes (such as role names).
- **method** The HTTP Method which will be used in combination with the pattern and servlet path

(optional) to match an incoming request. If omitted, any method will match. If an identical pattern is specified with and without a method, the method-specific match will take precedence.

- **pattern** The pattern which defines the URL path. The content will depend on the `request-matcher` attribute from the containing http element, so will default to ant path syntax.
- **request-matcher-ref** A reference to a `RequestMatcher` that will be used to determine if this `<intercept-url>` is used.
- **requires-channel** Can be "http" or "https" depending on whether a particular URL pattern should be accessed over HTTP or HTTPS respectively. Alternatively the value "any" can be used when there is no preference. If this attribute is present on any `<intercept-url>` element, then a `ChannelProcessingFilter` will be added to the filter stack and its additional dependencies added to the application context.

If a `<port-mappings>` configuration is added, this will be used to by the `SecureChannelProcessor` and `InsecureChannelProcessor` beans to determine the ports used for redirecting to HTTP/HTTPS.

This property is invalid for filter-security-metadata-source

- **servlet-path** The servlet path which will be used in combination with the pattern and HTTP method to match an incoming request. This attribute is only applicable when request-matcher is 'mvc'. In addition, the value is only required in the following 2 use cases: 1) There are 2 or more `HttpServlet` 's registered in the `ServletContext` that have mappings starting with `'/'` and are different; 2) The pattern starts with the same value of a registered `HttpServlet` path, excluding the default (root) `HttpServlet` `'/'`.

This property is invalid for filter-security-metadata-source

<jee>

Adds a J2eePreAuthenticatedProcessingFilter to the filter chain to provide integration with container authentication.

Parent Elements of <jee>

- http

<jee> Attributes

- **mappable-roles** A comma-separate list of roles to look for in the incoming HttpServletRequest.
- **user-service-ref** A reference to a user-service (or UserDetailsService bean) Id

<logout>

Adds a `LogoutFilter` to the filter stack. This is configured with a `SecurityContextLogoutHandler`.

Parent Elements of <logout>

- http

<logout> Attributes

- **delete-cookies** A comma-separated list of the names of cookies which should be deleted when the user logs out.
- **invalidate-session** Maps to the `invalidateHttpSession` of the `SecurityContextLogoutHandler`. Defaults to "true", so the session will be invalidated on logout.
- **logout-success-url** The destination URL which the user will be taken to after logging out. Defaults to <form-login-login-page>/?logout (i.e. /login?logout)

  Setting this attribute will inject the `SessionManagementFilter` with a `SimpleRedirectInvalidSessionStrategy` configured with the attribute value. When an invalid session ID is submitted, the strategy will be invoked, redirecting to the configured URL.

- **logout-url** The URL which will cause a logout (i.e. which will be processed by the filter). Defaults to "/logout".
- **success-handler-ref** May be used to supply an instance of `LogoutSuccessHandler` which will be invoked to control the navigation after logging out.

<openid-login>

Similar to `<form-login>` and has the same attributes. The default value for `login-processing-url` is "/login/openid". An `OpenIDAuthenticationFilter` and `OpenIDAuthenticationProvider` will be registered. The latter requires a reference to a `UserDetailsService`. Again, this can be specified by `id`, using the `user-service-ref` attribute, or will be located automatically in the application context.

Parent Elements of <openid-login>

- http

<openid-login> Attributes

- **always-use-default-target** Whether the user should always be redirected to the default-target-url after login.
- **authentication-details-source-ref** Reference to an AuthenticationDetailsSource which will be used by the authentication filter
- **authentication-failure-handler-ref** Reference to an AuthenticationFailureHandler bean which should be used to handle a failed authentication request. Should not be used in combination with authentication-failure-url as the implementation should always deal with navigation to the subsequent destination
- **authentication-failure-url** The URL for the login failure page. If no login failure URL is specified, Spring Security will automatically create a failure login URL at /login?login_error and

a corresponding filter to render that login failure URL when requested.

- **authentication-success-forward-url** Maps a `ForwardAuthenticationSuccessHandler` to `authenticationSuccessHandler` property of `UsernamePasswordAuthenticationFilter`.
- **authentication-failure-forward-url** Maps a `ForwardAuthenticationFailureHandler` to `authenticationFailureHandler` property of `UsernamePasswordAuthenticationFilter`.
- **authentication-success-handler-ref** Reference to an AuthenticationSuccessHandler bean which should be used to handle a successful authentication request. Should not be used in combination with default-target-url (or always-use-default-target) as the implementation should always deal with navigation to the subsequent destination
- **default-target-url** The URL that will be redirected to after successful authentication, if the user's previous action could not be resumed. This generally happens if the user visits a login page without having first requested a secured operation that triggers authentication. If unspecified, defaults to the root of the application.
- **login-page** The URL for the login page. If no login URL is specified, Spring Security will automatically create a login URL at /login and a corresponding filter to render that login URL when requested.
- **login-processing-url** The URL that the login form is posted to. If unspecified, it defaults to /login.
- **password-parameter** The name of the request parameter which contains the password. Defaults to "password".
- **user-service-ref** A reference to a user-service (or UserDetailsService bean) Id
- **username-parameter** The name of the request parameter which contains the username. Defaults to "username".

Child Elements of <openid-login>

- attribute-exchange

<attribute-exchange>

The `attribute-exchange` element defines the list of attributes which should be requested from the identity provider. An example can be found in the OpenID Support section of the namespace configuration chapter. More than one can be used, in which case each must have an `identifier-match` attribute, containing a regular expression which is matched against the supplied OpenID identifier. This allows different attribute lists to be fetched from different providers (Google, Yahoo etc).

Parent Elements of <attribute-exchange>

- openid-login

<attribute-exchange> Attributes

- **identifier-match** A regular expression which will be compared against the claimed identity, when deciding which attribute-exchange configuration to use during authentication.

Child Elements of <attribute-exchange>

- openid-attribute

<openid-attribute>

Attributes used when making an OpenID AX Fetch Request

Parent Elements of <openid-attribute>

- attribute-exchange

<openid-attribute> Attributes

- **count** Specifies the number of attributes that you wish to get back. For example, return 3 emails. The default value is 1.
- **name** Specifies the name of the attribute that you wish to get back. For example, email.
- **required** Specifies if this attribute is required to the OP, but does not error out if the OP does not return the attribute. Default is false.
- **type** Specifies the attribute type. For example, https://axschema.org/contact/email. See your OP's documentation for valid attribute types.

<password-management>

This element configures password management.

Parent Elements of <password-management>

- http

<password-management> Attributes

- **change-password-page** The change password page. Defaults to "/change-password".

<port-mappings>

By default, an instance of `PortMapperImpl` will be added to the configuration for use in redirecting to secure and insecure URLs. This element can optionally be used to override the default mappings which that class defines. Each child `<port-mapping>` element defines a pair of HTTP:HTTPS ports. The default mappings are 80:443 and 8080:8443. An example of overriding these can be found in Redirect to HTTPS.

Parent Elements of <port-mappings>

- http

Child Elements of <port-mappings>

- port-mapping

<port-mapping>

Provides a method to map http ports to https ports when forcing a redirect.

Parent Elements of <port-mapping>

- port-mappings

<port-mapping> Attributes

- **http** The http port to use.
- **https** The https port to use.

<remember-me>

Adds the `RememberMeAuthenticationFilter` to the stack. This in turn will be configured with either a `TokenBasedRememberMeServices`, a `PersistentTokenBasedRememberMeServices` or a user-specified bean implementing `RememberMeServices` depending on the attribute settings.

Parent Elements of <remember-me>

- http

<remember-me> Attributes

- **authentication-success-handler-ref** Sets the `authenticationSuccessHandler` property on the `RememberMeAuthenticationFilter` if custom navigation is required. The value should be the name of a `AuthenticationSuccessHandler` bean in the application context.
- **data-source-ref** A reference to a `DataSource` bean. If this is set, `PersistentTokenBasedRememberMeServices` will be used and configured with a `JdbcTokenRepositoryImpl` instance.
- **remember-me-parameter** The name of the request parameter which toggles remember-me authentication. Defaults to "remember-me". Maps to the "parameter" property of `AbstractRememberMeServices`.
- **remember-me-cookie** The name of cookie which store the token for remember-me authentication. Defaults to "remember-me". Maps to the "cookieName" property of `AbstractRememberMeServices`.
- **key** Maps to the "key" property of `AbstractRememberMeServices`. Should be set to a unique value to ensure that remember-me cookies are only valid within the one application [15]. If this is not set a secure random value will be generated. Since generating secure random values can take a while, setting this value explicitly can help improve startup times when using the remember-me functionality.

- **services-alias** Exports the internally defined `RememberMeServices` as a bean alias, allowing it to be used by other beans in the application context.
- **services-ref** Allows complete control of the `RememberMeServices` implementation that will be used by the filter. The value should be the `id` of a bean in the application context which implements this interface. Should also implement `LogoutHandler` if a logout filter is in use.
- **token-repository-ref** Configures a `PersistentTokenBasedRememberMeServices` but allows the use of a custom `PersistentTokenRepository` bean.
- **token-validity-seconds** Maps to the `tokenValiditySeconds` property of `AbstractRememberMeServices`. Specifies the period in seconds for which the remember-me cookie should be valid. By default it will be valid for 14 days.
- **use-secure-cookie** It is recommended that remember-me cookies are only submitted over HTTPS and thus should be flagged as "secure". By default, a secure cookie will be used if the connection over which the login request is made is secure (as it should be). If you set this property to `false`, secure cookies will not be used. Setting it to `true` will always set the secure flag on the cookie. This attribute maps to the `useSecureCookie` property of `AbstractRememberMeServices`.
- **user-service-ref** The remember-me services implementations require access to a `UserDetailsService`, so there has to be one defined in the application context. If there is only one, it will be selected and used automatically by the namespace configuration. If there are multiple instances, you can specify a bean `id` explicitly using this attribute.

<request-cache> Element

Sets the `RequestCache` instance which will be used by the `ExceptionTranslationFilter` to store request information before invoking an `AuthenticationEntryPoint`.

Parent Elements of <request-cache>

- http

<request-cache> Attributes

- **ref** Defines a reference to a Spring bean that is a `RequestCache`.

<session-management>

Session-management related functionality is implemented by the addition of a `SessionManagementFilter` to the filter stack.

Parent Elements of <session-management>

- http

<session-management> Attributes

- **invalid-session-url** Setting this attribute will inject the `SessionManagementFilter` with a

`SimpleRedirectInvalidSessionStrategy` configured with the attribute value. When an invalid session ID is submitted, the strategy will be invoked, redirecting to the configured URL.

- **invalid-session-url** Allows injection of the InvalidSessionStrategy instance used by the SessionManagementFilter. Use either this or the `invalid-session-url` attribute but not both.
- **session-authentication-error-url** Defines the URL of the error page which should be shown when the SessionAuthenticationStrategy raises an exception. If not set, an unauthorized (401) error code will be returned to the client. Note that this attribute doesn't apply if the error occurs during a form-based login, where the URL for authentication failure will take precedence.
- **session-authentication-strategy-ref** Allows injection of the SessionAuthenticationStrategy instance used by the SessionManagementFilter
- **session-fixation-protection** Indicates how session fixation protection will be applied when a user authenticates. If set to "none", no protection will be applied. "newSession" will create a new empty session, with only Spring Security-related attributes migrated. "migrateSession" will create a new session and copy all session attributes to the new session. In Servlet 3.1 (Java EE 7) and newer containers, specifying "changeSessionId" will keep the existing session and use the container-supplied session fixation protection (HttpServletRequest#changeSessionId()). Defaults to "changeSessionId" in Servlet 3.1 and newer containers, "migrateSession" in older containers. Throws an exception if "changeSessionId" is used in older containers.

  If session fixation protection is enabled, the `SessionManagementFilter` is injected with an appropriately configured `DefaultSessionAuthenticationStrategy`. See the Javadoc for this class for more details.

Child Elements of <session-management>

- concurrency-control

<concurrency-control>

Adds support for concurrent session control, allowing limits to be placed on the number of active sessions a user can have. A `ConcurrentSessionFilter` will be created, and a `ConcurrentSessionControlAuthenticationStrategy` will be used with the `SessionManagementFilter`. If a `form-login` element has been declared, the strategy object will also be injected into the created authentication filter. An instance of `SessionRegistry` (a `SessionRegistryImpl` instance unless the user wishes to use a custom bean) will be created for use by the strategy.

Parent Elements of <concurrency-control>

- session-management

<concurrency-control> Attributes

- **error-if-maximum-exceeded** If set to "true" a `SessionAuthenticationException` will be raised when a user attempts to exceed the maximum allowed number of sessions. The default behaviour is to expire the original session.

- **expired-url** The URL a user will be redirected to if they attempt to use a session which has been "expired" by the concurrent session controller because the user has exceeded the number of allowed sessions and has logged in again elsewhere. Should be set unless `exception-if-maximum-exceeded` is set. If no value is supplied, an expiry message will just be written directly back to the response.
- **expired-url** Allows injection of the ExpiredSessionStrategy instance used by the ConcurrentSessionFilter
- **max-sessions** Maps to the `maximumSessions` property of `ConcurrentSessionControlAuthenticationStrategy`. Specify `-1` as the value to support unlimited sessions.
- **session-registry-alias** It can also be useful to have a reference to the internal session registry for use in your own beans or an admin interface. You can expose the internal bean using the `session-registry-alias` attribute, giving it a name that you can use elsewhere in your configuration.
- **session-registry-ref** The user can supply their own `SessionRegistry` implementation using the `session-registry-ref` attribute. The other concurrent session control beans will be wired up to use it.

<x509>

Adds support for X.509 authentication. An `X509AuthenticationFilter` will be added to the stack and an `Http403ForbiddenEntryPoint` bean will be created. The latter will only be used if no other authentication mechanisms are in use (its only functionality is to return an HTTP 403 error code). A `PreAuthenticatedAuthenticationProvider` will also be created which delegates the loading of user authorities to a `UserDetailsService`.

Parent Elements of <x509>

- http

<x509> Attributes

- **authentication-details-source-ref** A reference to an `AuthenticationDetailsSource`
- **subject-principal-regex** Defines a regular expression which will be used to extract the username from the certificate (for use with the `UserDetailsService`).
- **user-service-ref** Allows a specific `UserDetailsService` to be used with X.509 in the case where multiple instances are configured. If not set, an attempt will be made to locate a suitable instance automatically and use that.

<filter-chain-map>

Used to explicitly configure a FilterChainProxy instance with a FilterChainMap

<filter-chain-map> Attributes

- **request-matcher** Defines the strategy to use for matching incoming requests. Currently the options are 'ant' (for ant path patterns), 'regex' for regular expressions and 'ciRegex' for case-insensitive regular expressions.

Child Elements of <filter-chain-map>

- filter-chain

<filter-chain>

Used within to define a specific URL pattern and the list of filters which apply to the URLs matching that pattern. When multiple filter-chain elements are assembled in a list in order to configure a FilterChainProxy, the most specific patterns must be placed at the top of the list, with most general ones at the bottom.

Parent Elements of <filter-chain>

- filter-chain-map

<filter-chain> Attributes

- **filters** A comma separated list of references to Spring beans that implement `Filter`. The value "none" means that no `Filter` should be used for this `FilterChain`.
- **pattern** A pattern that creates RequestMatcher in combination with the request-matcher
- **request-matcher-ref** A reference to a `RequestMatcher` that will be used to determine if any `Filter` from the `filters` attribute should be invoked.

<filter-security-metadata-source>

Used to explicitly configure a FilterSecurityMetadataSource bean for use with a FilterSecurityInterceptor. Usually only needed if you are configuring a FilterChainProxy explicitly, rather than using the<http> element. The intercept-url elements used should only contain pattern, method and access attributes. Any others will result in a configuration error.

<filter-security-metadata-source> Attributes

- **id** A bean identifier, used for referring to the bean elsewhere in the context.
- **request-matcher** Defines the strategy use for matching incoming requests. Currently the options are 'ant' (for ant path patterns), 'regex' for regular expressions and 'ciRegex' for case-insensitive regular expressions.
- **use-expressions** Enables the use of expressions in the 'access' attributes in <intercept-url> elements rather than the traditional list of configuration attributes. Defaults to 'true'. If enabled, each attribute should contain a single Boolean expression. If the expression evaluates to 'true', access will be granted.

Child Elements of <filter-security-metadata-source>

- intercept-url

21.2.2. WebSocket Security

Spring Security 4.0+ provides support for authorizing messages. One concrete example of where this is useful is to provide authorization in WebSocket based applications.

<websocket-message-broker>

The websocket-message-broker element has two different modes. If the websocket-message-broker@id is not specified, then it will do the following things:

- Ensure that any SimpAnnotationMethodMessageHandler has the AuthenticationPrincipalArgumentResolver registered as a custom argument resolver. This allows the use of `@AuthenticationPrincipal` to resolve the principal of the current `Authentication`
- Ensures that the SecurityContextChannelInterceptor is automatically registered for the clientInboundChannel. This populates the SecurityContextHolder with the user that is found in the Message
- Ensures that a ChannelSecurityInterceptor is registered with the clientInboundChannel. This allows authorization rules to be specified for a message.
- Ensures that a CsrfChannelInterceptor is registered with the clientInboundChannel. This ensures that only requests from the original domain are enabled.
- Ensures that a CsrfTokenHandshakeInterceptor is registered with WebSocketHttpRequestHandler, TransportHandlingSockJsService, or DefaultSockJsService. This ensures that the expected CsrfToken from the HttpServletRequest is copied into the WebSocket Session attributes.

If additional control is necessary, the id can be specified and a ChannelSecurityInterceptor will be assigned to the specified id. All the wiring with Spring's messaging infrastructure can then be done manually. This is more cumbersome, but provides greater control over the configuration.

<websocket-message-broker> Attributes

- **id** A bean identifier, used for referring to the ChannelSecurityInterceptor bean elsewhere in the context. If specified, Spring Security requires explicit configuration within Spring Messaging. If not specified, Spring Security will automatically integrate with the messaging infrastructure as described in <websocket-message-broker>
- **same-origin-disabled** Disables the requirement for CSRF token to be present in the Stomp headers (default false). Changing the default is useful if it is necessary to allow other origins to make SockJS connections.

Child Elements of <websocket-message-broker>

- expression-handler
- intercept-message

<intercept-message>

Defines an authorization rule for a message.

Parent Elements of <intercept-message>

- websocket-message-broker

<intercept-message> Attributes

- **pattern** An ant based pattern that matches on the Message destination. For example, "/" **matches any Message with a destination; "/admin/**" matches any Message that has a destination that starts with "/admin/**".
- **type** The type of message to match on. Valid values are defined in SimpMessageType (i.e. CONNECT, CONNECT_ACK, HEARTBEAT, MESSAGE, SUBSCRIBE, UNSUBSCRIBE, DISCONNECT, DISCONNECT_ACK, OTHER).
- **access** The expression used to secure the Message. For example, "denyAll" will deny access to all of the matching Messages; "permitAll" will grant access to all of the matching Messages; "hasRole('ADMIN') requires the current user to have the role 'ROLE_ADMIN' for the matching Messages.

21.2.3. Authentication Services

Before Spring Security 3.0, an `AuthenticationManager` was automatically registered internally. Now you must register one explicitly using the `<authentication-manager>` element. This creates an instance of Spring Security's `ProviderManager` class, which needs to be configured with a list of one or more `AuthenticationProvider` instances. These can either be created using syntax elements provided by the namespace, or they can be standard bean definitions, marked for addition to the list using the `authentication-provider` element.

<authentication-manager>

Every Spring Security application which uses the namespace must have include this element somewhere. It is responsible for registering the `AuthenticationManager` which provides authentication services to the application. All elements which create `AuthenticationProvider` instances should be children of this element.

<authentication-manager> Attributes

- **alias** This attribute allows you to define an alias name for the internal instance for use in your own configuration.
- **erase-credentials** If set to true, the AuthenticationManager will attempt to clear any credentials data in the returned Authentication object, once the user has been authenticated. Literally it maps to the `eraseCredentialsAfterAuthentication` property of the `ProviderManager`.
- **id** This attribute allows you to define an id for the internal instance for use in your own configuration. It is the same as the alias element, but provides a more consistent experience with elements that use the id attribute.

Child Elements of <authentication-manager>

- authentication-provider
- ldap-authentication-provider

<authentication-provider>

Unless used with a `ref` attribute, this element is shorthand for configuring a `DaoAuthenticationProvider`. `DaoAuthenticationProvider` loads user information from a `UserDetailsService` and compares the username/password combination with the values supplied at login. The `UserDetailsService` instance can be defined either by using an available namespace element (`jdbc-user-service` or by using the `user-service-ref` attribute to point to a bean defined elsewhere in the application context).

Parent Elements of <authentication-provider>

- authentication-manager

<authentication-provider> Attributes

- **ref** Defines a reference to a Spring bean that implements `AuthenticationProvider`.

If you have written your own `AuthenticationProvider` implementation (or want to configure one of Spring Security's own implementations as a traditional bean for some reason, then you can use the following syntax to add it to the internal list of `ProviderManager`:

```
<security:authentication-manager>
<security:authentication-provider ref="myAuthenticationProvider" />
</security:authentication-manager>
<bean id="myAuthenticationProvider" class="com.something.MyAuthenticationProvider"/>
```

- **user-service-ref** A reference to a bean that implements UserDetailsService that may be created using the standard bean element or the custom user-service element.

Child Elements of <authentication-provider>

- jdbc-user-service
- ldap-user-service
- password-encoder
- user-service

<jdbc-user-service>

Causes creation of a JDBC-based UserDetailsService.

<jdbc-user-service> Attributes

- **authorities-by-username-query** An SQL statement to query for a user's granted authorities given a username.

The default is

```
select username, authority from authorities where username = ?
```

- **cache-ref** Defines a reference to a cache for use with a UserDetailsService.
- **data-source-ref** The bean ID of the DataSource which provides the required tables.
- **group-authorities-by-username-query** An SQL statement to query user's group authorities given a username. The default is

  ```
  select
  g.id, g.group_name, ga.authority
  from
  groups g, group_members gm, group_authorities ga
  where
  gm.username = ? and g.id = ga.group_id and g.id = gm.group_id
  ```

- **id** A bean identifier, used for referring to the bean elsewhere in the context.
- **role-prefix** A non-empty string prefix that will be added to role strings loaded from persistent storage (default is "ROLE_"). Use the value "none" for no prefix in cases where the default is non-empty.
- **users-by-username-query** An SQL statement to query a username, password, and enabled status given a username. The default is

  ```
  select username, password, enabled from users where username = ?
  ```

<password-encoder>

Authentication providers can optionally be configured to use a password encoder as described in the Password Storage. This will result in the bean being injected with the appropriate `PasswordEncoder` instance.

Parent Elements of <password-encoder>

- authentication-provider
- password-compare

<password-encoder> Attributes

- **hash** Defines the hashing algorithm used on user passwords. We recommend strongly against using MD4, as it is a very weak hashing algorithm.
- **ref** Defines a reference to a Spring bean that implements `PasswordEncoder`.

<user-service>

Creates an in-memory UserDetailsService from a properties file or a list of "user" child elements. Usernames are converted to lower-case internally to allow for case-insensitive lookups, so this should not be used if case-sensitivity is required.

<user-service> Attributes

- **id** A bean identifier, used for referring to the bean elsewhere in the context.
- **properties** The location of a Properties file where each line is in the format of

```
username=password,grantedAuthority[,grantedAuthority][,enabled|disabled]
```

Child Elements of <user-service>

- user

<user>

Represents a user in the application.

Parent Elements of <user>

- user-service

<user> Attributes

- **authorities** One of more authorities granted to the user. Separate authorities with a comma (but no space). For example, "ROLE_USER,ROLE_ADMINISTRATOR"
- **disabled** Can be set to "true" to mark an account as disabled and unusable.
- **locked** Can be set to "true" to mark an account as locked and unusable.
- **name** The username assigned to the user.
- **password** The password assigned to the user. This may be hashed if the corresponding authentication provider supports hashing (remember to set the "hash" attribute of the "user-service" element). This attribute be omitted in the case where the data will not be used for authentication, but only for accessing authorities. If omitted, the namespace will generate a random value, preventing its accidental use for authentication. Cannot be empty.

21.2.4. Method Security

<method-security>

This element is the primary means of adding support for securing methods on Spring Security beans. Methods can be secured by the use of annotations (defined at the interface or class level) or by defining a set of pointcuts.

<method-security> attributes

- **pre-post-enabled** Enables Spring Security's pre and post invocation annotations (@PreFilter, @PreAuthorize, @PostFilter, @PostAuthorize) for this application context. Defaults to "true".
- **secured-enabled** Enables Spring Security's @Secured annotation for this application context. Defaults to "false".
- **jsr250-enabled** Enables JSR-250 authorization annotations (@RolesAllowed, @PermitAll, @DenyAll) for this application context. Defaults to "false".
- **proxy-target-class** If true, class based proxying will be used instead of interface based proxying. Defaults to "false".

Child Elements of <method-security>

- expression-handler

<global-method-security>

This element is the primary means of adding support for securing methods on Spring Security beans. Methods can be secured by the use of annotations (defined at the interface or class level) or by defining a set of pointcuts as child elements, using AspectJ syntax.

<global-method-security> Attributes

- **access-decision-manager-ref** Method security uses the same `AccessDecisionManager` configuration as web security, but this can be overridden using this attribute. By default an AffirmativeBased implementation is used for with a RoleVoter and an AuthenticatedVoter.
- **authentication-manager-ref** A reference to an `AuthenticationManager` that should be used for method security.
- **jsr250-annotations** Specifies whether JSR-250 style attributes are to be used (for example "RolesAllowed"). This will require the javax.annotation.security classes on the classpath. Setting this to true also adds a `Jsr250Voter` to the `AccessDecisionManager`, so you need to make sure you do this if you are using a custom implementation and want to use these annotations.
- **metadata-source-ref** An external `MethodSecurityMetadataSource` instance can be supplied which will take priority over other sources (such as the default annotations).
- **mode** This attribute can be set to "aspectj" to specify that AspectJ should be used instead of the default Spring AOP. Secured methods must be woven with the `AnnotationSecurityAspect` from the `spring-security-aspects` module.

It is important to note that AspectJ follows Java's rule that annotations on interfaces are not inherited. This means that methods that define the Security annotations on the interface will not be secured. Instead, you must place the Security annotation on the class when using AspectJ.

- **order** Allows the advice "order" to be set for the method security interceptor.
- **pre-post-annotations** Specifies whether the use of Spring Security's pre and post invocation

annotations (@PreFilter, @PreAuthorize, @PostFilter, @PostAuthorize) should be enabled for this application context. Defaults to "disabled".

- **proxy-target-class** If true, class based proxying will be used instead of interface based proxying.
- **run-as-manager-ref** A reference to an optional `RunAsManager` implementation which will be used by the configured `MethodSecurityInterceptor`
- **secured-annotations** Specifies whether the use of Spring Security's @Secured annotations should be enabled for this application context. Defaults to "disabled".

Child Elements of <global-method-security>

- after-invocation-provider
- expression-handler
- pre-post-annotation-handling
- protect-pointcut

<after-invocation-provider>

This element can be used to decorate an `AfterInvocationProvider` for use by the security interceptor maintained by the `<global-method-security>` namespace. You can define zero or more of these within the `global-method-security` element, each with a `ref` attribute pointing to an `AfterInvocationProvider` bean instance within your application context.

Parent Elements of <after-invocation-provider>

- global-method-security

<after-invocation-provider> Attributes

- **ref** Defines a reference to a Spring bean that implements `AfterInvocationProvider`.

<pre-post-annotation-handling>

Allows the default expression-based mechanism for handling Spring Security's pre and post invocation annotations (@PreFilter, @PreAuthorize, @PostFilter, @PostAuthorize) to be replaced entirely. Only applies if these annotations are enabled.

Parent Elements of <pre-post-annotation-handling>

- global-method-security

Child Elements of <pre-post-annotation-handling>

- invocation-attribute-factory
- post-invocation-advice
- pre-invocation-advice

<invocation-attribute-factory>

Defines the PrePostInvocationAttributeFactory instance which is used to generate pre and post invocation metadata from the annotated methods.

Parent Elements of <invocation-attribute-factory>

- pre-post-annotation-handling

<invocation-attribute-factory> Attributes

- **ref** Defines a reference to a Spring bean Id.

<post-invocation-advice>

Customizes the `PostInvocationAdviceProvider` with the ref as the `PostInvocationAuthorizationAdvice` for the <pre-post-annotation-handling> element.

Parent Elements of <post-invocation-advice>

- pre-post-annotation-handling

<post-invocation-advice> Attributes

- **ref** Defines a reference to a Spring bean Id.

<pre-invocation-advice>

Customizes the `PreInvocationAuthorizationAdviceVoter` with the ref as the `PreInvocationAuthorizationAdviceVoter` for the <pre-post-annotation-handling> element.

Parent Elements of <pre-invocation-advice>

- pre-post-annotation-handling

<pre-invocation-advice> Attributes

- **ref** Defines a reference to a Spring bean Id.

Securing Methods using

`<protect-pointcut>` Rather than defining security attributes on an individual method or class basis using the `@Secured` annotation, you can define cross-cutting security constraints across whole sets of methods and interfaces in your service layer using the `<protect-pointcut>` element. You can find an example in the namespace introduction.

Parent Elements of <protect-pointcut>

- global-method-security

<protect-pointcut> Attributes

- **access** Access configuration attributes list that applies to all methods matching the pointcut, e.g.

"ROLE_A,ROLE_B"

- **expression** An AspectJ expression, including the `execution` keyword. For example, `execution(int com.foo.TargetObject.countLength(String))`.

<intercept-methods>

Can be used inside a bean definition to add a security interceptor to the bean and set up access configuration attributes for the bean's methods

<intercept-methods> Attributes

- **access-decision-manager-ref** Optional AccessDecisionManager bean ID to be used by the created method security interceptor.

Child Elements of <intercept-methods>

- protect

<method-security-metadata-source>

Creates a MethodSecurityMetadataSource instance

<method-security-metadata-source> Attributes

- **id** A bean identifier, used for referring to the bean elsewhere in the context.
- **use-expressions** Enables the use of expressions in the 'access' attributes in <intercept-url> elements rather than the traditional list of configuration attributes. Defaults to 'false'. If enabled, each attribute should contain a single Boolean expression. If the expression evaluates to 'true', access will be granted.

Child Elements of <method-security-metadata-source>

- protect

<protect>

Defines a protected method and the access control configuration attributes that apply to it. We strongly advise you NOT to mix "protect" declarations with any services provided "global-method-security".

Parent Elements of <protect>

- intercept-methods
- method-security-metadata-source

<protect> Attributes

- **access** Access configuration attributes list that applies to the method, e.g. "ROLE_A,ROLE_B".
- **method** A method name

21.2.5. LDAP Namespace Options

LDAP is covered in some details in its own chapter. We will expand on that here with some explanation of how the namespace options map to Spring beans. The LDAP implementation uses Spring LDAP extensively, so some familiarity with that project's API may be useful.

Defining the LDAP Server using the

`<ldap-server>` Element This element sets up a Spring LDAP `ContextSource` for use by the other LDAP beans, defining the location of the LDAP server and other information (such as a username and password, if it doesn't allow anonymous access) for connecting to it. It can also be used to create an embedded server for testing. Details of the syntax for both options are covered in the LDAP chapter. The actual `ContextSource` implementation is `DefaultSpringSecurityContextSource` which extends Spring LDAP's `LdapContextSource` class. The `manager-dn` and `manager-password` attributes map to the latter's `userDn` and `password` properties respectively.

If you only have one server defined in your application context, the other LDAP namespace-defined beans will use it automatically. Otherwise, you can give the element an "id" attribute and refer to it from other namespace beans using the `server-ref` attribute. This is actually the bean `id` of the `ContextSource` instance, if you want to use it in other traditional Spring beans.

<ldap-server> Attributes

- **mode** Explicitly specifies which embedded ldap server should use. Values are `apacheds` and `unboundid`. By default, it will depends if the library is available in the classpath.
- **id** A bean identifier, used for referring to the bean elsewhere in the context.
- **ldif** Explicitly specifies an ldif file resource to load into an embedded LDAP server. The ldif should be a Spring resource pattern (i.e. classpath:init.ldif). The default is classpath*:*.ldif
- **manager-dn** Username (DN) of the "manager" user identity which will be used to authenticate to a (non-embedded) LDAP server. If omitted, anonymous access will be used.
- **manager-password** The password for the manager DN. This is required if the manager-dn is specified.
- **port** Specifies an IP port number. Used to configure an embedded LDAP server, for example. The default value is 33389.
- **root** Optional root suffix for the embedded LDAP server. Default is "dc=springframework,dc=org"
- **url** Specifies the ldap server URL when not using the embedded LDAP server.

<ldap-authentication-provider>

This element is shorthand for the creation of an `LdapAuthenticationProvider` instance. By default this will be configured with a `BindAuthenticator` instance and a `DefaultAuthoritiesPopulator`. As with all namespace authentication providers, it must be included as a child of the `authentication-provider` element.

Parent Elements of <ldap-authentication-provider>

- authentication-manager

<ldap-authentication-provider> Attributes

- **group-role-attribute** The LDAP attribute name which contains the role name which will be used within Spring Security. Maps to the `DefaultLdapAuthoritiesPopulator`'s `groupRoleAttribute` property. Defaults to "cn".
- **group-search-base** Search base for group membership searches. Maps to the `DefaultLdapAuthoritiesPopulator`'s `groupSearchBase` constructor argument. Defaults to "" (searching from the root).
- **group-search-filter** Group search filter. Maps to the `DefaultLdapAuthoritiesPopulator`'s `groupSearchFilter` property. Defaults to `(uniqueMember={0})`. The substituted parameter is the DN of the user.
- **role-prefix** A non-empty string prefix that will be added to role strings loaded from persistent. Maps to the `DefaultLdapAuthoritiesPopulator`'s `rolePrefix` property. Defaults to "ROLE_". Use the value "none" for no prefix in cases where the default is non-empty.
- **server-ref** The optional server to use. If omitted, and a default LDAP server is registered (using <ldap-server> with no Id), that server will be used.
- **user-context-mapper-ref** Allows explicit customization of the loaded user object by specifying a UserDetailsContextMapper bean which will be called with the context information from the user's directory entry
- **user-details-class** Allows the objectClass of the user entry to be specified. If set, the framework will attempt to load standard attributes for the defined class into the returned UserDetails object
- **user-dn-pattern** If your users are at a fixed location in the directory (i.e. you can work out the DN directly from the username without doing a directory search), you can use this attribute to map directly to the DN. It maps directly to the `userDnPatterns` property of `AbstractLdapAuthenticator`. The value is a specific pattern used to build the user's DN, for example `uid={0},ou=people`. The key `{0}` must be present and will be substituted with the username.
- **user-search-base** Search base for user searches. Defaults to "". Only used with a 'user-search-filter'.

  If you need to perform a search to locate the user in the directory, then you can set these attributes to control the search. The `BindAuthenticator` will be configured with a `FilterBasedLdapUserSearch` and the attribute values map directly to the first two arguments of that bean's constructor. If these attributes aren't set and no `user-dn-pattern` has been supplied as an alternative, then the default search values of `user-search-filter="(uid={0})"` and `user-search-base=""` will be used.

- **user-search-filter** The LDAP filter used to search for users (optional). For example `(uid={0})`.

The substituted parameter is the user's login name.

If you need to perform a search to locate the user in the directory, then you can set these attributes to control the search. The `BindAuthenticator` will be configured with a `FilterBasedLdapUserSearch` and the attribute values map directly to the first two arguments of that bean's constructor. If these attributes aren't set and no `user-dn-pattern` has been supplied as an alternative, then the default search values of `user-search-filter="(uid={0})"` and `user-search-base=""` will be used.

Child Elements of <ldap-authentication-provider>

- password-compare

<password-compare>

This is used as child element to `<ldap-provider>` and switches the authentication strategy from `BindAuthenticator` to `PasswordComparisonAuthenticator`.

Parent Elements of <password-compare>

- ldap-authentication-provider

<password-compare> Attributes

- **hash** Defines the hashing algorithm used on user passwords. We recommend strongly against using MD4, as it is a very weak hashing algorithm.
- **password-attribute** The attribute in the directory which contains the user password. Defaults to "userPassword".

Child Elements of <password-compare>

- password-encoder

<ldap-user-service>

This element configures an LDAP `UserDetailsService`. The class used is `LdapUserDetailsService` which is a combination of a `FilterBasedLdapUserSearch` and a `DefaultLdapAuthoritiesPopulator`. The attributes it supports have the same usage as in `<ldap-provider>`.

<ldap-user-service> Attributes

- **cache-ref** Defines a reference to a cache for use with a UserDetailsService.
- **group-role-attribute** The LDAP attribute name which contains the role name which will be used within Spring Security. Defaults to "cn".
- **group-search-base** Search base for group membership searches. Defaults to "" (searching from the root).
- **group-search-filter** Group search filter. Defaults to `(uniqueMember={0})`. The substituted

parameter is the DN of the user.

- **id** A bean identifier, used for referring to the bean elsewhere in the context.
- **role-prefix** A non-empty string prefix that will be added to role strings loaded from persistent storage (e.g. "ROLE_"). Use the value "none" for no prefix in cases where the default is non-empty.
- **server-ref** The optional server to use. If omitted, and a default LDAP server is registered (using <ldap-server> with no Id), that server will be used.
- **user-context-mapper-ref** Allows explicit customization of the loaded user object by specifying a UserDetailsContextMapper bean which will be called with the context information from the user's directory entry
- **user-details-class** Allows the objectClass of the user entry to be specified. If set, the framework will attempt to load standard attributes for the defined class into the returned UserDetails object
- **user-search-base** Search base for user searches. Defaults to "". Only used with a 'user-search-filter'.
- **user-search-filter** The LDAP filter used to search for users (optional). For example `(uid={0})`. The substituted parameter is the user's login name.

21.3. Spring Security FAQ

21.3.1. General Questions

Will Spring Security take care of all my application security requirements?

Spring Security provides you with a very flexible framework for your authentication and authorization requirements, but there are many other considerations for building a secure application that are outside its scope. Web applications are vulnerable to all kinds of attacks which

you should be familiar with, preferably before you start development so you can design and code with them in mind from the beginning. Check out the OWASP web site for information on the major issues facing web application developers and the countermeasures you can use against them.

Why not just use web.xml security?

Let's assume you're developing an enterprise application based on Spring. There are four security concerns you typically need to address: authentication, web request security, service layer security (i.e. your methods that implement business logic), and domain object instance security (i.e. different domain objects have different permissions). With these typical requirements in mind:

1. *Authentication*: The servlet specification provides an approach to authentication. However, you will need to configure the container to perform authentication which typically requires editing of container-specific "realm" settings. This makes a non-portable configuration, and if you need to write an actual Java class to implement the container's authentication interface, it becomes even more non-portable. With Spring Security you achieve complete portability - right down to the WAR level. Also, Spring Security offers a choice of production-proven authentication providers and mechanisms, meaning you can switch your authentication approaches at deployment time. This is particularly valuable for software vendors writing products that need to work in an unknown target environment.
2. *Web request security:* The servlet specification provides an approach to secure your request URIs. However, these URIs can only be expressed in the servlet specification's own limited URI path format. Spring Security provides a far more comprehensive approach. For instance, you can use Ant paths or regular expressions, you can consider parts of the URI other than simply the requested page (e.g. you can consider HTTP GET parameters) and you can implement your own runtime source of configuration data. This means your web request security can be dynamically changed during the actual execution of your webapp.
3. *Service layer and domain object security:* The absence of support in the servlet specification for services layer security or domain object instance security represent serious limitations for multi-tiered applications. Typically developers either ignore these requirements, or implement security logic within their MVC controller code (or even worse, inside the views). There are serious disadvantages with this approach:
    a. *Separation of concerns:* Authorization is a crosscutting concern and should be implemented as such. MVC controllers or views implementing authorization code makes it more difficult to test both the controller and authorization logic, more difficult to debug, and will often lead to code duplication.
    b. *Support for rich clients and web services:* If an additional client type must ultimately be supported, any authorization code embedded within the web layer is non-reusable. It should be considered that Spring remoting exporters only export service layer beans (not MVC controllers). As such authorization logic needs to be located in the services layer to support a multitude of client types.
    c. *Layering issues:* An MVC controller or view is simply the incorrect architectural layer to implement authorization decisions concerning services layer methods or domain object instances. Whilst the Principal may be passed to the services layer to enable it to make the authorization decision, doing so would introduce an additional argument on every services layer method. A more elegant approach is to use a ThreadLocal to hold the Principal,

although this would likely increase development time to a point where it would become more economical (on a cost-benefit basis) to simply use a dedicated security framework.

d. *Authorisation code quality:* It is often said of web frameworks that they "make it easier to do the right things, and harder to do the wrong things". Security frameworks are the same, because they are designed in an abstract manner for a wide range of purposes. Writing your own authorization code from scratch does not provide the "design check" a framework would offer, and in-house authorization code will typically lack the improvements that emerge from widespread deployment, peer review and new versions.

For simple applications, servlet specification security may just be enough. Although when considered within the context of web container portability, configuration requirements, limited web request security flexibility, and non-existent services layer and domain object instance security, it becomes clear why developers often look to alternative solutions.

What Java and Spring Framework versions are required?

Spring Security 3.0 and 3.1 require at least JDK 1.5 and also require Spring 3.0.3 as a minimum. Ideally you should be using the latest release versions to avoid problems.

Spring Security 2.0.x requires a minimum JDK version of 1.4 and is built against Spring 2.0.x. It should also be compatible with applications using Spring 2.5.x.

I'm new to Spring Security and I need to build an application that supports CAS single sign-on over HTTPS, while allowing Basic authentication locally for certain URLs, authenticating against multiple back end user information sources (LDAP and JDBC). I've copied some configuration files I found but it doesn't work.

What could be wrong?

Or substitute an alternative complex scenario…

Realistically, you need an understanding of the technologies you are intending to use before you can successfully build applications with them. Security is complicated. Setting up a simple configuration using a login form and some hard-coded users using Spring Security's namespace is reasonably straightforward. Moving to using a backed JDBC database is also easy enough. But if you try and jump straight to a complicated deployment scenario like this you will almost certainly be frustrated. There is a big jump in the learning curve required to set up systems like CAS, configure LDAP servers and install SSL certificates properly. So you need to take things one step at a time.

From a Spring Security perspective, the first thing you should do is follow the "Getting Started" guide on the web site. This will take you through a series of steps to get up and running and get some idea of how the framework operates. If you are using other technologies which you aren't familiar with then you should do some research and try to make sure you can use them in isolation before combining them in a complex system.

21.3.2. Common Problems

1. Authentication
    a. When I try to log in, I get an error message that says "Bad Credentials". What's wrong?

When I try to log in, I get an error message that says "Bad Credentials". What's wrong?

This means that authentication has failed. It doesn't say why, as it is good practice to avoid giving details which might help an attacker guess account names or passwords.

This also means that if you ask this question in the forum, you will not get an answer unless you provide additional information. As with any issue you should check the output from the debug log, note any exception stacktraces and related messages. Step through the code in a debugger to see where the authentication fails and why. Write a test case which exercises your authentication configuration outside of the application. More often than not, the failure is due to a difference in the password data stored in a database and that entered by the user. If you are using hashed passwords, make sure the value stored in your database is *exactly* the same as the value produced by the `PasswordEncoder` configured in your application.

My application goes into an "endless loop" when I try to login, what's going on?

A common user problem with infinite loop and redirecting to the login page is caused by accidentally configuring the login page as a "secured" resource. Make sure your configuration allows anonymous access to the login page, either by excluding it from the security filter chain or marking it as requiring ROLE_ANONYMOUS.

If your AccessDecisionManager includes an AuthenticatedVoter, you can use the attribute "IS_AUTHENTICATED_ANONYMOUSLY". This is automatically available if you are using the standard namespace configuration setup.

From Spring Security 2.0.1 onwards, when you are using namespace-based configuration, a check will be made on loading the application context and a warning message logged if your login page appears to be protected.

I get an exception with the message "Access is denied (user is anonymous);". What's wrong?

This is a debug level message which occurs the first time an anonymous user attempts to access a protected resource.

```
DEBUG [ExceptionTranslationFilter] - Access is denied (user is anonymous); redirecting
to authentication entry point
org.springframework.security.AccessDeniedException: Access is denied
at org.springframework.security.vote.AffirmativeBased.decide(AffirmativeBased.java:68)
at
org.springframework.security.intercept.AbstractSecurityInterceptor.beforeInvocation(Ab
stractSecurityInterceptor.java:262)
```

It is normal and shouldn't be anything to worry about.

Why can I still see a secured page even after I've logged out of my application?

The most common reason for this is that your browser has cached the page and you are seeing a copy which is being retrieved from the browsers cache. Verify this by checking whether the browser is actually sending the request (check your server access logs, the debug log or use a suitable browser debugging plugin such as "Tamper Data" for Firefox). This has nothing to do with Spring Security and you should configure your application or server to set the appropriate `Cache-Control` response headers. Note that SSL requests are never cached.

I get an exception with the message "An Authentication object was not found in the SecurityContext". What's wrong?

This is a another debug level message which occurs the first time an anonymous user attempts to access a protected resource, but when you do not have an `AnonymousAuthenticationFilter` in your filter chain configuration.

```
DEBUG [ExceptionTranslationFilter] - Authentication exception occurred; redirecting to
authentication entry point
org.springframework.security.AuthenticationCredentialsNotFoundException:
                            An Authentication object was not found in the
SecurityContext
at
org.springframework.security.intercept.AbstractSecurityInterceptor.credentialsNotFound
(AbstractSecurityInterceptor.java:342)
at
org.springframework.security.intercept.AbstractSecurityInterceptor.beforeInvocation(Ab
stractSecurityInterceptor.java:254)
```

It is normal and shouldn't be anything to worry about.

I can't get LDAP authentication to work.

What's wrong with my configuration?

Note that the permissions for an LDAP directory often do not allow you to read the password for a user. Hence it is often not possible to use the What is a UserDetailsService and do I need one? where Spring Security compares the stored password with the one submitted by the user. The most common approach is to use LDAP "bind", which is one of the operations supported by the LDAP protocol. With this approach, Spring Security validates the password by attempting to authenticate to the directory as the user.

The most common problem with LDAP authentication is a lack of knowledge of the directory server tree structure and configuration. This will be different in different companies, so you have to find it out yourself. Before adding a Spring Security LDAP configuration to an application, it's a good idea to write a simple test using standard Java LDAP code (without Spring Security involved), and make sure you can get that to work first. For example, to authenticate a user, you could use the following code:

Java

```
@Test
public void ldapAuthenticationIsSuccessful() throws Exception {
        Hashtable<String,String> env = new Hashtable<String,String>();
        env.put(Context.SECURITY_AUTHENTICATION, "simple");
        env.put(Context.SECURITY_PRINCIPAL,
"cn=joe,ou=users,dc=mycompany,dc=com");
        env.put(Context.PROVIDER_URL,
"ldap://mycompany.com:389/dc=mycompany,dc=com");
        env.put(Context.SECURITY_CREDENTIALS, "joespassword");
        env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.ldap.LdapCtxFactory");

        InitialLdapContext ctx = new InitialLdapContext(env, null);

}
```

Kotlin

```
@Test
fun ldapAuthenticationIsSuccessful() {
    val env = Hashtable<String, String>()
    env[Context.SECURITY_AUTHENTICATION] = "simple"
    env[Context.SECURITY_PRINCIPAL] = "cn=joe,ou=users,dc=mycompany,dc=com"
    env[Context.PROVIDER_URL] = "ldap://mycompany.com:389/dc=mycompany,dc=com"
    env[Context.SECURITY_CREDENTIALS] = "joespassword"
    env[Context.INITIAL_CONTEXT_FACTORY] = "com.sun.jndi.ldap.LdapCtxFactory"
    val ctx = InitialLdapContext(env, null)
}
```

Session Management

Session management issues are a common source of forum questions. If you are developing Java web applications, you should understand how the session is maintained between the servlet container and the user's browser. You should also understand the difference between secure and non-secure cookies and the implications of using HTTP/HTTPS and switching between the two. Spring Security has nothing to do with maintaining the session or providing session identifiers. This is entirely handled by the servlet container.

I'm using Spring Security's concurrent session control to prevent users from logging in more than once at a time.

When I open another browser window after logging in, it doesn't stop me from logging in again. Why can I log in more than once?

Browsers generally maintain a single session per browser instance. You cannot have two separate sessions at once. So if you log in again in another window or tab you are just reauthenticating in

the same session. The server doesn't know anything about tabs, windows or browser instances. All it sees are HTTP requests and it ties those to a particular session according to the value of the JSESSIONID cookie that they contain. When a user authenticates during a session, Spring Security's concurrent session control checks the number of *other authenticated sessions* that they have. If they are already authenticated with the same session, then re-authenticating will have no effect.

Why does the session Id change when I authenticate through Spring Security?

With the default configuration, Spring Security changes the session ID when the user authenticates. If you're using a Servlet 3.1 or newer container, the session ID is simply changed. If you're using an older container, Spring Security invalidates the existing session, creates a new session, and transfers the session data to the new session. Changing the session identifier in this manner prevents"session-fixation" attacks. You can find more about this online and in the reference manual.

I'm using Tomcat (or some other servlet container) and have enabled HTTPS for my login page, switching back to HTTP afterwards.

It doesn't work - I just end up back at the login page after authenticating.

This happens because sessions created under HTTPS, for which the session cookie is marked as "secure", cannot subsequently be used under HTTP. The browser will not send the cookie back to the server and any session state will be lost (including the security context information). Starting a session in HTTP first should work as the session cookie won't be marked as secure. However, Spring Security's Session Fixation Protection can interfere with this because it results in a new session ID cookie being sent back to the user's browser, usually with the secure flag. To get around this, you can disable session fixation protection, but in newer Servlet containers you can also configure session cookies to never use the secure flag. Note that switching between HTTP and HTTPS is not a good idea in general, as any application which uses HTTP at all is vulnerable to man-in-the-middle attacks. To be truly secure, the user should begin accessing your site in HTTPS and continue using it until they log out. Even clicking on an HTTPS link from a page accessed over HTTP is potentially risky. If you need more convincing, check out a tool like sslstrip.

I'm not switching between HTTP and HTTPS but my session is still getting lost

Sessions are maintained either by exchanging a session cookie or by adding a `jsessionid` parameter to URLs (this happens automatically if you are using JSTL to output URLs, or if you call `HttpServletResponse.encodeUrl` on URLs (before a redirect, for example). If clients have cookies disabled, and you are not rewriting URLs to include the `jsessionid`, then the session will be lost. Note that the use of cookies is preferred for security reasons, as it does not expose the session information in the URL.

I'm trying to use the concurrent session-control support but it won't let me log back in, even if I'm sure I've logged out and haven't exceeded the allowed sessions.

Make sure you have added the listener to your web.xml file. It is essential to make sure that the Spring Security session registry is notified when a session is destroyed. Without it, the session information will not be removed from the registry.

```
<listener>
        <listener-
class>org.springframework.security.web.session.HttpSessionEventPublisher</listener-
class>
</listener>
```

Spring Security is creating a session somewhere, even though I've configured it not to, by setting the create-session attribute to never.

This usually means that the user's application is creating a session somewhere, but that they aren't aware of it. The most common culprit is a JSP. Many people aren't aware that JSPs create sessions by default. To prevent a JSP from creating a session, add the directive `<%@ page session="false" %>` to the top of the page.

If you are having trouble working out where a session is being created, you can add some debugging code to track down the location(s). One way to do this would be to add a `javax.servlet.http.HttpSessionListener` to your application, which calls `Thread.dumpStack()` in the `sessionCreated` method.

I get a 403 Forbidden when performing a POST

If an HTTP 403 Forbidden is returned for HTTP POST, but works for HTTP GET then the issue is most likely related to CSRF. Either provide the CSRF Token or disable CSRF protection (not recommended).

I'm forwarding a request to another URL using the RequestDispatcher, but my security constraints aren't being applied.

Filters are not applied by default to forwards or includes. If you really want the security filters to be applied to forwards and/or includes, then you have to configure these explicitly in your web.xml using the <dispatcher> element, a child element of <filter-mapping>.

I have added Spring Security's <global-method-security> element to my application context but if I add security annotations to my Spring MVC controller beans (Struts actions etc.) then they don't seem to have an effect.

In a Spring web application, the application context which holds the Spring MVC beans for the dispatcher servlet is often separate from the main application context. It is often defined in a file called `myapp-servlet.xml`, where "myapp" is the name assigned to the Spring `DispatcherServlet` in `web.xml`. An application can have multiple `DispatcherServlet`s, each with its own isolated application context. The beans in these "child" contexts are not visible to the rest of the application. The"parent" application context is loaded by the `ContextLoaderListener` you define in your `web.xml` and is visible to all the child contexts. This parent context is usually where you define your security configuration, including the `<global-method-security>` element). As a result any security constraints applied to methods in these web beans will not be enforced, since the beans cannot be seen from the `DispatcherServlet` context. You need to either move the `<global-method-security>` declaration to the web context or moved the beans you want secured into the main application context.

Generally we would recommend applying method security at the service layer rather than on

individual web controllers.

I have a user who has definitely been authenticated, but when I try to access the SecurityContextHolder during some requests, the Authentication is null.

Why can't I see the user information?

If you have excluded the request from the security filter chain using the attribute `filters='none'` in the `<intercept-url>` element that matches the URL pattern, then the `SecurityContextHolder` will not be populated for that request. Check the debug log to see whether the request is passing through the filter chain. (You are reading the debug log, right?).

The authorize JSP Tag doesn't respect my method security annotations when using the URL attribute.

Method security will not hide links when using the `url` attribute in `<sec:authorize>` because we cannot readily reverse engineer what URL is mapped to what controller endpoint as controllers can rely on headers, current user, etc to determine what method to invoke.

21.3.3. Spring Security Architecture Questions

1. How do I know which package class X is in?
2. How do the namespace elements map to conventional bean configurations?
3. What does "ROLE_" mean and why do I need it on my role names?
4. How do I know which dependencies to add to my application to work with Spring Security?
5. What dependencies are needed to run an embedded ApacheDS LDAP server?
6. What is a UserDetailsService and do I need one?

How do I know which package class X is in?

The best way of locating classes is by installing the Spring Security source in your IDE. The distribution includes source jars for each of the modules the project is divided up into. Add these to your project source path and you can navigate directly to Spring Security classes (`Ctrl-Shift-T` in Eclipse). This also makes debugging easier and allows you to troubleshoot exceptions by looking directly at the code where they occur to see what's going on there.

How do the namespace elements map to conventional bean configurations?

There is a general overview of what beans are created by the namespace in the namespace appendix of the reference guide. There is also a detailed blog article called "Behind the Spring Security Namespace" on blog.springsource.com. If want to know the full details then the code is in the `spring-security-config` module within the Spring Security 3.0 distribution. You should probably read the chapters on namespace parsing in the standard Spring Framework reference documentation first.

What does "ROLE_" mean and why do I need it on my role names?

Spring Security has a voter-based architecture which means that an access decision is made by a

series of AccessDecisionVoters. The voters act on the "configuration attributes" which are specified for a secured resource (such as a method invocation). With this approach, not all attributes may be relevant to all voters and a voter needs to know when it should ignore an attribute (abstain) and when it should vote to grant or deny access based on the attribute value. The most common voter is the RoleVoter which by default votes whenever it finds an attribute with the "ROLE_" prefix. It makes a simple comparison of the attribute (such as "ROLE_USER") with the names of the authorities which the current user has been assigned. If it finds a match (they have an authority called "ROLE_USER"), it votes to grant access, otherwise it votes to deny access.

The prefix can be changed by setting the rolePrefix property of RoleVoter. If you only need to use roles in your application and have no need for other custom voters, then you can set the prefix to a blank string, in which case the RoleVoter will treat all attributes as roles.

How do I know which dependencies to add to my application to work with Spring Security?

It will depend on what features you are using and what type of application you are developing. With Spring Security 3.0, the project jars are divided into clearly distinct areas of functionality, so it is straightforward to work out which Spring Security jars you need from your application requirements. All applications will need the spring-security-core jar. If you're developing a web application, you need the spring-security-web jar. If you're using security namespace configuration you need the spring-security-config jar, for LDAP support you need the spring-security-ldap jar and so on.

For third-party jars the situation isn't always quite so obvious. A good starting point is to copy those from one of the pre-built sample applications WEB-INF/lib directories. For a basic application, you can start with the tutorial sample. If you want to use LDAP, with an embedded test server, then use the LDAP sample as a starting point. The reference manual also includes an appendix listing the first-level dependencies for each Spring Security module with some information on whether they are optional and what they are required for.

If you are building your project with maven, then adding the appropriate Spring Security modules as dependencies to your pom.xml will automatically pull in the core jars that the framework requires. Any which are marked as "optional" in the Spring Security POM files will have to be added to your own pom.xml file if you need them.

What dependencies are needed to run an embedded ApacheDS LDAP server?

If you are using Maven, you need to add the following to your pom dependencies:

```
<dependency>
        <groupId>org.apache.directory.server</groupId>
        <artifactId>apacheds-core</artifactId>
        <version>1.5.5</version>
        <scope>runtime</scope>
</dependency>
<dependency>
        <groupId>org.apache.directory.server</groupId>
        <artifactId>apacheds-server-jndi</artifactId>
        <version>1.5.5</version>
        <scope>runtime</scope>
</dependency>
```

The other required jars should be pulled in transitively.

What is a UserDetailsService and do I need one?

`UserDetailsService` is a DAO interface for loading data that is specific to a user account. It has no other function other to load that data for use by other components within the framework. It is not responsible for authenticating the user. Authenticating a user with a username/password combination is most commonly performed by the `DaoAuthenticationProvider`, which is injected with a `UserDetailsService` to allow it to load the password (and other data) for a user in order to compare it with the submitted value. Note that if you are using LDAP, this approach may not work.

If you want to customize the authentication process then you should implement `AuthenticationProvider` yourself. See this blog article for an example integrating Spring Security authentication with Google App Engine.

21.3.4. Common "Howto" Requests

1. I need to login in with more information than just the username.
2. How do I apply different intercept-url constraints where only the fragment value of the requested URLs differs (e.g./foo#bar and /foo#blah?
3. How do I access the user's IP Address (or other web-request data) in a UserDetailsService?
4. How do I access the HttpSession from a UserDetailsService?
5. How do I access the user's password in a UserDetailsService?
6. How do I define the secured URLs within an application dynamically?
7. How do I authenticate against LDAP but load user roles from a database?
8. I want to modify the property of a bean that is created by the namespace, but there is nothing in the schema to support it.

I need to login in with more information than just the username.

How do I add support for extra login fields (e.g. a company name)?

This question comes up repeatedly in the Spring Security forum so you will find more information

there by searching the archives (or through google).

The submitted login information is processed by an instance of `UsernamePasswordAuthenticationFilter`. You will need to customize this class to handle the extra data field(s). One option is to use your own customized authentication token class (rather than the standard `UsernamePasswordAuthenticationToken`), another is simply to concatenate the extra fields with the username (for example, using a ":" as the separator) and pass them in the username property of `UsernamePasswordAuthenticationToken`.

You will also need to customize the actual authentication process. If you are using a custom authentication token class, for example, you will have to write an `AuthenticationProvider` to handle it (or extend the standard `DaoAuthenticationProvider`). If you have concatenated the fields, you can implement your own `UserDetailsService` which splits them up and loads the appropriate user data for authentication.

How do I apply different intercept-url constraints where only the fragment value of the requested URLs differs (e.g./foo#bar and /foo#blah?

You can't do this, since the fragment is not transmitted from the browser to the server. The URLs above are identical from the server's perspective. This is a common question from GWT users.

How do I access the user's IP Address (or other web-request data) in a UserDetailsService?

Obviously you can't (without resorting to something like thread-local variables) since the only information supplied to the interface is the username. Instead of implementing `UserDetailsService`, you should implement `AuthenticationProvider` directly and extract the information from the supplied `Authentication` token.

In a standard web setup, the `getDetails()` method on the `Authentication` object will return an instance of `WebAuthenticationDetails`. If you need additional information, you can inject a custom `AuthenticationDetailsSource` into the authentication filter you are using. If you are using the namespace, for example with the `<form-login>` element, then you should remove this element and replace it with a `<custom-filter>` declaration pointing to an explicitly configured `UsernamePasswordAuthenticationFilter`.

How do I access the HttpSession from a UserDetailsService?

You can't, since the `UserDetailsService` has no awareness of the servlet API. If you want to store custom user data, then you should customize the `UserDetails` object which is returned. This can then be accessed at any point, via the thread-local `SecurityContextHolder`. A call to `SecurityContextHolder.getContext().getAuthentication().getPrincipal()` will return this custom object.

If you really need to access the session, then it must be done by customizing the web tier.

How do I access the user's password in a UserDetailsService?

You can't (and shouldn't). You are probably misunderstanding its purpose. See "What is a UserDetailsService?" above.

How do I define the secured URLs within an application dynamically?

People often ask about how to store the mapping between secured URLs and security metadata attributes in a database, rather than in the application context.

The first thing you should ask yourself is if you really need to do this. If an application requires securing, then it also requires that the security be tested thoroughly based on a defined policy. It may require auditing and acceptance testing before being rolled out into a production environment. A security-conscious organization should be aware that the benefits of their diligent testing process could be wiped out instantly by allowing the security settings to be modified at runtime by changing a row or two in a configuration database. If you have taken this into account (perhaps using multiple layers of security within your application) then Spring Security allows you to fully customize the source of security metadata. You can make it fully dynamic if you choose.

Both method and web security are protected by subclasses of `AbstractSecurityInterceptor` which is configured with a `SecurityMetadataSource` from which it obtains the metadata for a particular method or filter invocation. For web security, the interceptor class is `FilterSecurityInterceptor` and it uses the marker interface `FilterInvocationSecurityMetadataSource`. The "secured object" type it operates on is a `FilterInvocation`. The default implementation which is used (both in the namespace `<http>` and when configuring the interceptor explicitly, stores the list of URL patterns and their corresponding list of "configuration attributes" (instances of `ConfigAttribute`) in an in-memory map.

To load the data from an alternative source, you must be using an explicitly declared security filter chain (typically Spring Security's `FilterChainProxy`) in order to customize the `FilterSecurityInterceptor` bean. You can't use the namespace. You would then implement `FilterInvocationSecurityMetadataSource` to load the data as you please for a particular `FilterInvocation` [16]. A very basic outline would look something like this:

Java

```
public class MyFilterSecurityMetadataSource implements
FilterInvocationSecurityMetadataSource {

    public List<ConfigAttribute> getAttributes(Object object) {
        FilterInvocation fi = (FilterInvocation) object;
            String url = fi.getRequestUrl();
            String httpMethod = fi.getRequest().getMethod();
            List<ConfigAttribute> attributes = new ArrayList<ConfigAttribute>();

            // Lookup your database (or other source) using this information and
populate the
            // list of attributes

            return attributes;
    }

    public Collection<ConfigAttribute> getAllConfigAttributes() {
        return null;
    }

    public boolean supports(Class<?> clazz) {
        return FilterInvocation.class.isAssignableFrom(clazz);
    }
}
```

Kotlin

```
class MyFilterSecurityMetadataSource : FilterInvocationSecurityMetadataSource {
    override fun getAttributes(securedObject: Any): List<ConfigAttribute> {
        val fi = securedObject as FilterInvocation
        val url = fi.requestUrl
        val httpMethod = fi.request.method

        // Lookup your database (or other source) using this information and
populate the
        // list of attributes
        return ArrayList()
    }

    override fun getAllConfigAttributes(): Collection<ConfigAttribute>? {
        return null
    }

    override fun supports(clazz: Class<*>): Boolean {
        return FilterInvocation::class.java.isAssignableFrom(clazz)
    }
}
```

For more information, look at the code for `DefaultFilterInvocationSecurityMetadataSource`.

How do I authenticate against LDAP but load user roles from a database?

The `LdapAuthenticationProvider` bean (which handles normal LDAP authentication in Spring Security) is configured with two separate strategy interfaces, one which performs the authentication and one which loads the user authorities, called `LdapAuthenticator` and `LdapAuthoritiesPopulator` respectively. The `DefaultLdapAuthoritiesPopulator` loads the user authorities from the LDAP directory and has various configuration parameters to allow you to specify how these should be retrieved.

To use JDBC instead, you can implement the interface yourself, using whatever SQL is appropriate for your schema:

Java

```
public class MyAuthoritiesPopulator implements LdapAuthoritiesPopulator {
    @Autowired
    JdbcTemplate template;

    List<GrantedAuthority> getGrantedAuthorities(DirContextOperations userData,
String username) {
        return template.query("select role from roles where username = ?",
                new String[] {username},
                new RowMapper<GrantedAuthority>() {
             /**
              *  We're assuming here that you're using the standard convention of
using the role
              *  prefix "ROLE_" to mark attributes which are supported by Spring
Security's RoleVoter.
              */
             @Override
             public GrantedAuthority mapRow(ResultSet rs, int rowNum) throws
SQLException {
                 return new SimpleGrantedAuthority("ROLE_" + rs.getString(1));
             }
        });
    }
}
```

Kotlin

```
class MyAuthoritiesPopulator : LdapAuthoritiesPopulator {
    @Autowired
    lateinit var template: JdbcTemplate

    override fun getGrantedAuthorities(userData: DirContextOperations, username:
String): MutableList<GrantedAuthority?> {
        return template.query("select role from roles where username = ?",
            arrayOf(username)
        ) { rs, _ ->
            /**
             * We're assuming here that you're using the standard convention of
using the role
             * prefix "ROLE_" to mark attributes which are supported by Spring
Security's RoleVoter.
             */
            SimpleGrantedAuthority("ROLE_" + rs.getString(1))
        }
    }
}
```

You would then add a bean of this type to your application context and inject it into the `LdapAuthenticationProvider`. This is covered in the section on configuring LDAP using explicit Spring beans in the LDAP chapter of the reference manual. Note that you can't use the namespace for configuration in this case. You should also consult the Javadoc for the relevant classes and interfaces.

I want to modify the property of a bean that is created by the namespace, but there is nothing in the schema to support it.

What can I do short of abandoning namespace use?

The namespace functionality is intentionally limited, so it doesn't cover everything that you can do with plain beans. If you want to do something simple, like modify a bean, or inject a different dependency, you can do this by adding a `BeanPostProcessor` to your configuration. More information can be found in the Spring Reference Manual. In order to do this, you need to know a bit about which beans are created, so you should also read the blog article in the above question on how the namespace maps to Spring beans.

Normally, you would add the functionality you require to the `postProcessBeforeInitialization` method of `BeanPostProcessor`. Let's say that you want to customize the `AuthenticationDetailsSource` used by the `UsernamePasswordAuthenticationFilter`, (created by the `form-login` element). You want to extract a particular header called `CUSTOM_HEADER` from the request and make use of it while authenticating the user. The processor class would look like this:

Java

```
public class CustomBeanPostProcessor implements BeanPostProcessor {

        public Object postProcessAfterInitialization(Object bean, String name) {
                if (bean instanceof UsernamePasswordAuthenticationFilter) {
                        System.out.println("********* Post-processing " + name);

((UsernamePasswordAuthenticationFilter)bean).setAuthenticationDetailsSource(
                                        new AuthenticationDetailsSource() {
                                                public Object buildDetails(Object
context) {
                                                        return
((HttpServletRequest)context).getHeader("CUSTOM_HEADER");
                                                }
                                        });
                }
                return bean;
        }

        public Object postProcessBeforeInitialization(Object bean, String name) {
                return bean;
        }
}
```

Kotlin

```
class CustomBeanPostProcessor : BeanPostProcessor {
    override fun postProcessAfterInitialization(bean: Any, name: String): Any {
        if (bean is UsernamePasswordAuthenticationFilter) {
            println("********* Post-processing $name")
            bean.setAuthenticationDetailsSource(
                AuthenticationDetailsSource<HttpServletRequest, Any?> { context ->
context.getHeader("CUSTOM_HEADER") })
        }
        return bean
    }

    override fun postProcessBeforeInitialization(bean: Any, name: String?): Any {
        return bean
    }
}
```

You would then register this bean in your application context. Spring will automatically invoke it on the beans defined in the application context.

[13] See the introductory chapter for how to set up the mapping from your `web.xml`

[14] This feature is really just provided for convenience and is not intended for production (where a view technology will have been chosen and can be used to render a customized login page). The class `DefaultLoginPageGeneratingFilter` is responsible for

rendering the login page and will provide login forms for both normal form login and/or OpenID if required.

[15] This doesn't affect the use of `PersistentTokenBasedRememberMeServices`, where the tokens are stored on the server side.

[16] The `FilterInvocation` object contains the `HttpServletRequest`, so you can obtain the URL or any other relevant information on which to base your decision on what the list of returned attributes will contain.

Reactive Applications

Chapter 22. WebFlux Security

Spring Security's WebFlux support relies on a `WebFilter` and works the same for Spring WebFlux and Spring WebFlux.Fn. You can find a few sample applications that demonstrate the code below:

- Hello WebFlux hellowebflux
- Hello WebFlux.Fn hellowebfluxfn
- Hello WebFlux Method hellowebflux-method

22.1. Minimal WebFlux Security Configuration

You can find a minimal WebFlux Security configuration below:

Example 199. Minimal WebFlux Security Configuration

Java

```
@EnableWebFluxSecurity
public class HelloWebfluxSecurityConfig {

    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
            .roles("USER")
            .build();
        return new MapReactiveUserDetailsService(user);
    }
}
```

Kotlin

```
@EnableWebFluxSecurity
class HelloWebfluxSecurityConfig {

    @Bean
    fun userDetailsService(): ReactiveUserDetailsService {
        val userDetails = User.withDefaultPasswordEncoder()
                .username("user")
                .password("user")
                .roles("USER")
                .build()
        return MapReactiveUserDetailsService(userDetails)
    }
}
```

This configuration provides form and http basic authentication, sets up authorization to require an authenticated user for accessing any page, sets up a default log in page and a default log out page, sets up security related HTTP headers, CSRF protection, and more.

22.2. Explicit WebFlux Security Configuration

You can find an explicit version of the minimal WebFlux Security configuration below:

Example 200. Explicit WebFlux Security Configuration

Java

```
@Configuration
@EnableWebFluxSecurity
public class HelloWebfluxSecurityConfig {

    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
            .roles("USER")
            .build();
        return new MapReactiveUserDetailsService(user);
    }

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity
http) {
        http
            .authorizeExchange(exchanges -> exchanges
                .anyExchange().authenticated()
            )
            .httpBasic(withDefaults())
            .formLogin(withDefaults());
        return http.build();
    }
}
```

Kotlin

```
@Configuration
@EnableWebFluxSecurity
class HelloWebfluxSecurityConfig {

    @Bean
    fun userDetailsService(): ReactiveUserDetailsService {
        val userDetails = User.withDefaultPasswordEncoder()
                .username("user")
                .password("user")
                .roles("USER")
                .build()
        return MapReactiveUserDetailsService(userDetails)
    }

    @Bean
    fun springSecurityFilterChain(http: ServerHttpSecurity):
SecurityWebFilterChain {
        return http {
            authorizeExchange {
                authorize(anyExchange, authenticated)
            }
            formLogin { }
            httpBasic { }
        }
    }
}
```

This configuration explicitly sets up all the same things as our minimal configuration. From here you can easily make the changes to the defaults.

You can find more examples of explicit configuration in unit tests, by searching EnableWebFluxSecurity in the `config/src/test/` directory.

22.2.1. Multiple Chains Support

You can configure multiple `SecurityWebFilterChain` instances to separate configuration by `RequestMatcher` s.

For example, you can isolate configuration for URLs that start with `/api`, like so:

Java

```
@Configuration
@EnableWebFluxSecurity
static class MultiSecurityHttpConfig {

    @Order(Ordered.HIGHEST_PRECEDENCE)
①
    @Bean
    SecurityWebFilterChain apiHttpSecurity(ServerHttpSecurity http) {
        http
            .securityMatcher(new
PathPatternParserServerWebExchangeMatcher("/api/**"))      ②
            .authorizeExchange((exchanges) -> exchanges
                .anyExchange().authenticated()
            )
            .oauth2ResourceServer(OAuth2ResourceServerSpec::jwt);
③
        return http.build();
    }

    @Bean
    SecurityWebFilterChain webHttpSecurity(ServerHttpSecurity http) {
④
        http
            .authorizeExchange((exchanges) -> exchanges
                .anyExchange().authenticated()
            )
            .httpBasic(withDefaults());
⑤
        return http.build();
    }

    @Bean
    ReactiveUserDetailsService userDetailsService() {
        return new MapReactiveUserDetailsService(
                PasswordEncodedUser.user(), PasswordEncodedUser.admin());
    }

}
```

Kotlin

```
@Configuration
@EnableWebFluxSecurity
open class MultiSecurityHttpConfig {
    @Order(Ordered.HIGHEST_PRECEDENCE)
①
    @Bean
    open fun apiHttpSecurity(http: ServerHttpSecurity): SecurityWebFilterChain {
        return http {
            securityMatcher(PathPatternParserServerWebExchangeMatcher("/api/**"))
②
            authorizeExchange {
                authorize(anyExchange, authenticated)
            }
            oauth2ResourceServer {
                jwt { }
③
            }
        }
    }

    @Bean
    open fun webHttpSecurity(http: ServerHttpSecurity): SecurityWebFilterChain {
④
        return http {
            authorizeExchange {
                authorize(anyExchange, authenticated)
            }
            httpBasic { }
⑤
        }
    }

    @Bean
    open fun userDetailsService(): ReactiveUserDetailsService {
        return MapReactiveUserDetailsService(
            PasswordEncodedUser.user(), PasswordEncodedUser.admin()
        )
    }
}
```

① Configure a `SecurityWebFilterChain` with an `@Order` to specify which `SecurityWebFilterChain` Spring Security should consider first

② Use `PathPatternParserServerWebExchangeMatcher` to state that this `SecurityWebFilterChain` will only apply to URL paths that start with `/api/`

③ Specify the authentication mechanisms that will be used for `/api/**` endpoints

④ Create another instance of `SecurityWebFilterChain` with lower precedence to match all other

URLs

⑤ Specify the authentication mechanisms that will be used for the rest of the application

Spring Security will select one `SecurityWebFilterChain @Bean` for each request. It will match the requests in order by the `securityMatcher` definition.

In this case, that means that if the URL path starts with `/api`, then Spring Security will use `apiHttpSecurity`. If the URL does not start with `/api` then Spring Security will default to `webHttpSecurity`, which has an implied `securityMatcher` that matches any request.

Chapter 23. Protection Against Exploits

23.1. Cross Site Request Forgery (CSRF) for WebFlux Environments

This section discusses Spring Security's Cross Site Request Forgery (CSRF) support for WebFlux environments.

23.1.1. Using Spring Security CSRF Protection

The steps to using Spring Security's CSRF protection are outlined below:

- Use proper HTTP verbs
- Configure CSRF Protection
- Include the CSRF Token

Use proper HTTP verbs

The first step to protecting against CSRF attacks is to ensure your website uses proper HTTP verbs. This is covered in detail in Safe Methods Must be Idempotent.

Configure CSRF Protection

The next step is to configure Spring Security's CSRF protection within your application. Spring Security's CSRF protection is enabled by default, but you may need to customize the configuration. Below are a few common customizations.

Custom CsrfTokenRepository

By default Spring Security stores the expected CSRF token in the `WebSession` using `WebSessionServerCsrfTokenRepository`. There can be cases where users will want to configure a custom `ServerCsrfTokenRepository`. For example, it might be desirable to persist the `CsrfToken` in a cookie to support a JavaScript based application.

By default the `CookieServerCsrfTokenRepository` will write to a cookie named `XSRF-TOKEN` and read it from a header named `X-XSRF-TOKEN` or the HTTP parameter `_csrf`. These defaults come from AngularJS

You can configure `CookieServerCsrfTokenRepository` in Java Configuration using:

Example 201. Store CSRF Token in a Cookie

Java

```
@Bean
public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .csrf(csrf ->
csrf.csrfTokenRepository(CookieServerCsrfTokenRepository.withHttpOnlyFalse()))
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        csrf {
            csrfTokenRepository =
CookieServerCsrfTokenRepository.withHttpOnlyFalse()
        }
    }
}
```

The sample explicitly sets `cookieHttpOnly=false`. This is necessary to allow JavaScript (i.e. AngularJS) to read it. If you do not need the ability to read the cookie with JavaScript directly, it is recommended to omit `cookieHttpOnly=false` (by using `new CookieServerCsrfTokenRepository()` instead) to improve security.

Disable CSRF Protection

CSRF protection is enabled by default. However, it is simple to disable CSRF protection if it makes sense for your application.

The Java configuration below will disable CSRF protection.

Example 202. Disable CSRF Configuration

Java

```
@Bean
public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .csrf(csrf -> csrf.disable()))
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        csrf {
            disable()
        }
    }
}
```

Include the CSRF Token

In order for the synchronizer token pattern to protect against CSRF attacks, we must include the actual CSRF token in the HTTP request. This must be included in a part of the request (i.e. form parameter, HTTP header, etc) that is not automatically included in the HTTP request by the browser.

Spring Security's CsrfWebFilter exposes a Mono<CsrfToken> as a `ServerWebExchange` attribute named `org.springframework.security.web.server.csrf.CsrfToken`. This means that any view technology can access the `Mono<CsrfToken>` to expose the expected token as either a form or meta tag.

If your view technology does not provide a simple way to subscribe to the `Mono<CsrfToken>`, a common pattern is to use Spring's `@ControllerAdvice` to expose the `CsrfToken` directly. For example, the following code will place the `CsrfToken` on the default attribute name (`_csrf`) used by Spring Security's CsrfRequestDataValueProcessor to automatically include the CSRF token as a hidden input.

Example 203. `CsrfToken` as `@ModelAttribute`

Java

```
@ControllerAdvice
public class SecurityControllerAdvice {
    @ModelAttribute
    Mono<CsrfToken> csrfToken(ServerWebExchange exchange) {
        Mono<CsrfToken> csrfToken =
exchange.getAttribute(CsrfToken.class.getName());
        return csrfToken.doOnSuccess(token -> exchange.getAttributes()
                .put(CsrfRequestDataValueProcessor.DEFAULT_CSRF_ATTR_NAME,
token));
    }
}
```

Kotlin

```
@ControllerAdvice
class SecurityControllerAdvice {
    @ModelAttribute
    fun csrfToken(exchange: ServerWebExchange): Mono<CsrfToken> {
        val csrfToken: Mono<CsrfToken>? =
exchange.getAttribute(CsrfToken::class.java.name)
        return csrfToken!!.doOnSuccess { token ->

exchange.attributes[CsrfRequestDataValueProcessor.DEFAULT_CSRF_ATTR_NAME] = token
        }
    }
}
```

Fortunately, Thymeleaf provides integration that works without any additional work.

Form URL Encoded

In order to post an HTML form the CSRF token must be included in the form as a hidden input. For example, the rendered HTML might look like:

Example 204. CSRF Token HTML

```
<input type="hidden"
    name="_csrf"
    value="4bfd1575-3ad1-4d21-96c7-4ef2d9f86721"/>
```

Next we will discuss various ways of including the CSRF token in a form as a hidden input.

Automatic CSRF Token Inclusion

Spring Security's CSRF support provides integration with Spring's RequestDataValueProcessor via its CsrfRequestDataValueProcessor. In order for `CsrfRequestDataValueProcessor` to work, the `Mono<CsrfToken>` must be subscribed to and the `CsrfToken` must be exposed as an attribute that matches DEFAULT_CSRF_ATTR_NAME.

Fortunately, Thymeleaf provides support to take care of all the boilerplate for you by integrating with `RequestDataValueProcessor` to ensure that forms that have an unsafe HTTP method (i.e. post) will automatically include the actual CSRF token.

CsrfToken Request Attribute

If the other options for including the actual CSRF token in the request do not work, you can take advantage of the fact that the `Mono<CsrfToken>` is exposed as a `ServerWebExchange` attribute named `org.springframework.security.web.server.csrf.CsrfToken`.

The Thymeleaf sample below assumes that you expose the `CsrfToken` on an attribute named `_csrf`.

Example 205. CSRF Token in Form with Request Attribute

```
<form th:action="@{/logout}"
    method="post">
<input type="submit"
    value="Log out" />
<input type="hidden"
    th:name="${_csrf.parameterName}"
    th:value="${_csrf.token}"/>
</form>
```

Ajax and JSON Requests

If you are using JSON, then it is not possible to submit the CSRF token within an HTTP parameter. Instead you can submit the token within a HTTP header.

In the following sections we will discuss various ways of including the CSRF token as an HTTP request header in JavaScript based applications.

Automatic Inclusion

Spring Security can easily be configured to store the expected CSRF token in a cookie. By storing the expected CSRF in a cookie, JavaScript frameworks like AngularJS will automatically include the actual CSRF token in the HTTP request headers.

Meta tags

An alternative pattern to exposing the CSRF in a cookie is to include the CSRF token within your `meta` tags. The HTML might look something like this:

Example 206. CSRF meta tag HTML

```
<html>
<head>
    <meta name="_csrf" content="4bfd1575-3ad1-4d21-96c7-4ef2d9f86721"/>
    <meta name="_csrf_header" content="X-CSRF-TOKEN"/>
    <!-- ... -->
</head>
<!-- ... -->
```

Once the meta tags contained the CSRF token, the JavaScript code would read the meta tags and include the CSRF token as a header. If you were using jQuery, this could be done with the following:

Example 207. AJAX send CSRF Token

```
$(function () {
    var token = $("meta[name='_csrf']").attr("content");
    var header = $("meta[name='_csrf_header']").attr("content");
    $(document).ajaxSend(function(e, xhr, options) {
        xhr.setRequestHeader(header, token);
    });
});
```

The sample below assumes that you expose the `CsrfToken` on an attribute named `_csrf`. An example of doing this with Thymeleaf is shown below:

Example 208. CSRF meta tag JSP

```
<html>
<head>
    <meta name="_csrf" th:content="${_csrf.token}"/>
    <!-- default header name is X-CSRF-TOKEN -->
    <meta name="_csrf_header" th:content="${_csrf.headerName}"/>
    <!-- ... -->
</head>
<!-- ... -->
```

23.1.2. CSRF Considerations

There are a few special considerations to consider when implementing protection against CSRF attacks. This section discusses those considerations as it pertains to WebFlux environments. Refer to CSRF Considerations for a more general discussion.

Logging In

It is important to require CSRF for log in requests to protect against forging log in attempts. Spring Security's WebFlux support does this out of the box.

Logging Out

It is important to require CSRF for log out requests to protect against forging log out attempts. By default Spring Security's `LogoutWebFilter` only processes HTTP post requests. This ensures that log out requires a CSRF token and that a malicious user cannot forcibly log out your users.

The easiest approach is to use a form to log out. If you really want a link, you can use JavaScript to have the link perform a POST (i.e. maybe on a hidden form). For browsers with JavaScript that is disabled, you can optionally have the link take the user to a log out confirmation page that will perform the POST.

If you really want to use HTTP GET with logout you can do so, but remember this is generally not recommended. For example, the following Java Configuration will perform logout with the URL `/logout` is requested with any HTTP method:

Example 209. Log out with HTTP GET

Java

```
@Bean
public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .logout(logout -> logout.requiresLogout(new
PathPatternParserServerWebExchangeMatcher("/logout")))
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        logout {
            requiresLogout = PathPatternParserServerWebExchangeMatcher("/logout")
        }
    }
}
```

CSRF and Session Timeouts

By default Spring Security stores the CSRF token in the `WebSession`. This can lead to a situation where the session expires which means there is not an expected CSRF token to validate against.

We've already discussed general solutions to session timeouts. This section discusses the specifics of CSRF timeouts as it pertains to the WebFlux support.

It is simple to change storage of the expected CSRF token to be in a cookie. For details, refer to the Custom CsrfTokenRepository section.

Multipart (file upload)

We have already discussed how protecting multipart requests (file uploads) from CSRF attacks causes a chicken and the egg problem. This section discusses how to implement placing the CSRF token in the body and url within a WebFlux application.

More information about using multipart forms with Spring can be found within the Multipart Data section of the Spring reference.

Place CSRF Token in the Body

We have already discussed the trade-offs of placing the CSRF token in the body.

In a WebFlux application, this can be configured with the following configuration:

Example 210. Enable obtaining CSRF token from multipart/form-data

Java

```
@Bean
public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .csrf(csrf -> csrf.tokenFromMultipartDataEnabled(true))
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        csrf {
            tokenFromMultipartDataEnabled = true
        }
    }
}
```

Include CSRF Token in URL

We have already discussed the trade-offs of placing the CSRF token in the URL. Since the `CsrfToken` is exposed as an `ServerHttpRequest` request attribute, we can use that to create an `action` with the

CSRF token in it. An example with Thymeleaf is shown below:

Example 211. CSRF Token in Action

```
<form method="post"
    th:action="@{/upload(${_csrf.parameterName}=${_csrf.token})}"
    enctype="multipart/form-data">
```

HiddenHttpMethodFilter

We have already discussed overriding the HTTP method.

In a Spring WebFlux application, overriding the HTTP method is done using HiddenHttpMethodFilter.

23.2. Security HTTP Response Headers

Security HTTP Response Headers can be used to increase the security of web applications. This section is dedicated to WebFlux based support for Security HTTP Response Headers.

23.2.1. Default Security Headers

Spring Security provides a default set of Security HTTP Response Headers to provide secure defaults. While each of these headers are considered best practice, it should be noted that not all clients utilize the headers, so additional testing is encouraged.

You can customize specific headers. For example, assume that you want the defaults except you wish to specify `SAMEORIGIN` for X-Frame-Options.

You can easily do this with the following Configuration:

Example 212. Customize Default Security Headers

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .headers(headers -> headers
            .frameOptions(frameOptions -> frameOptions
                .mode(Mode.SAMEORIGIN)
            )
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        headers {
            frameOptions {
                mode = Mode.SAMEORIGIN
            }
        }
    }
}
```

If you do not want the defaults to be added and want explicit control over what should be used, you can disable the defaults. An example is provided below:

Example 213. Disable HTTP Security Response Headers

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .headers(headers -> headers.disable());
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        headers {
            disable()
        }
    }
}
```

23.2.2. Cache Control

Spring Security includes Cache Control headers by default.

However, if you actually want to cache specific responses, your application can selectively add them to the ServerHttpResponse to override the header set by Spring Security. This is useful to ensure things like CSS, JavaScript, and images are properly cached.

When using Spring WebFluxZz, this is typically done within your configuration. Details on how to do this can be found in the Static Resources portion of the Spring Reference documentation

If necessary, you can also disable Spring Security's cache control HTTP response headers.

Example 214. Cache Control Disabled

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .headers(headers -> headers
            .cache(cache -> cache.disable())
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        headers {
            cache {
                disable()
            }
        }
    }
}
```

23.2.3. Content Type Options

Spring Security includes Content-Type headers by default. However, you can disable it with:

Example 215. Content Type Options Disabled

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .headers(headers -> headers
            .contentTypeOptions(contentTypeOptions ->
contentTypeOptions.disable())
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        headers {
            contentTypeOptions {
                disable()
            }
        }
    }
}
```

23.2.4. HTTP Strict Transport Security (HSTS)

Spring Security provides the Strict Transport Security header by default. However, you can customize the results explicitly. For example, the following is an example of explicitly providing HSTS:

Example 216. Strict Transport Security

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .headers(headers -> headers
            .hsts(hsts -> hsts
                .includeSubdomains(true)
                .preload(true)
                .maxAge(Duration.ofDays(365))
            )
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        headers {
            hsts {
                includeSubdomains = true
                preload = true
                maxAge = Duration.ofDays(365)
            }
        }
    }
}
```

23.2.5. X-Frame-Options

By default, Spring Security disables rendering within an iframe using X-Frame-Options.

You can customize frame options to use the same origin using the following:

Example 217. X-Frame-Options: SAMEORIGIN

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .headers(headers -> headers
            .frameOptions(frameOptions -> frameOptions
                .mode(SAMEORIGIN)
            )
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        headers {
            frameOptions {
                mode = SAMEORIGIN
            }
        }
    }
}
```

23.2.6. X-XSS-Protection

By default, Spring Security instructs browsers to block reflected XSS attacks using the <<headers-xss-protection,X-XSS-Protection header>>. You can disable `X-XSS-Protection` with the following Configuration:

Example 218. X-XSS-Protection Customization

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .headers(headers -> headers
            .xssProtection(xssProtection -> xssProtection.disable())
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        headers {
            xssProtection {
                disable()
            }
        }
    }
}
```

23.2.7. Content Security Policy (CSP)

Spring Security does not add Content Security Policy by default, because a reasonable default is impossible to know without context of the application. The web application author must declare the security policy(s) to enforce and/or monitor for the protected resources.

For example, given the following security policy:

Example 219. Content Security Policy Example

```
Content-Security-Policy: script-src 'self' https://trustedscripts.example.com;
object-src https://trustedplugins.example.com; report-uri /csp-report-endpoint/
```

You can enable the CSP header as shown below:

Example 220. Content Security Policy

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .headers(headers -> headers
            .contentSecurityPolicy(policy -> policy
                .policyDirectives("script-src 'self'
https://trustedscripts.example.com; object-src https://trustedplugins.example.com;
report-uri /csp-report-endpoint/")
            )
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        headers {
            contentSecurityPolicy {
                policyDirectives = "script-src 'self'
https://trustedscripts.example.com; object-src https://trustedplugins.example.com;
report-uri /csp-report-endpoint/"
            }
        }
    }
}
```

To enable the CSP `report-only` header, provide the following configuration:

Example 221. Content Security Policy Report Only

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .headers(headers -> headers
            .contentSecurityPolicy(policy -> policy
                .policyDirectives("script-src 'self'
https://trustedscripts.example.com; object-src https://trustedplugins.example.com;
report-uri /csp-report-endpoint/")
                .reportOnly()
            )
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        headers {
            contentSecurityPolicy {
                policyDirectives = "script-src 'self'
https://trustedscripts.example.com; object-src https://trustedplugins.example.com;
report-uri /csp-report-endpoint/"
                reportOnly = true
            }
        }
    }
}
```

23.2.8. Referrer Policy

Spring Security does not add Referrer Policy headers by default. You can enable the Referrer Policy header using configuration as shown below:

Example 222. Referrer Policy Configuration

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .headers(headers -> headers
            .referrerPolicy(referrer -> referrer
                .policy(ReferrerPolicy.SAME_ORIGIN)
            )
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        headers {
            referrerPolicy {
                policy = ReferrerPolicy.SAME_ORIGIN
            }
        }
    }
}
```

23.2.9. Feature Policy

Spring Security does not add Feature Policy headers by default. The following `Feature-Policy` header:

Example 223. Feature-Policy Example

```
Feature-Policy: geolocation 'self'
```

You can enable the Feature Policy header as shown below:

Example 224. Feature-Policy Configuration

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .headers(headers -> headers
            .featurePolicy("geolocation 'self'")
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        headers {
            featurePolicy("geolocation 'self'")
        }
    }
}
```

23.2.10. Permissions Policy

Spring Security does not add Permissions Policy headers by default. The following `Permissions-Policy` header:

Example 225. Permissions-Policy Example

```
Permissions-Policy: geolocation=(self)
```

You can enable the Permissions Policy header as shown below:

Example 226. Permissions-Policy Configuration

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .headers(headers -> headers
            .permissionsPolicy(permissions -> permissions
                .policy("geolocation=(self)")
            )
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        headers {
            permissionsPolicy {
                policy = "geolocation=(self)"
            }
        }
    }
}
```

23.2.11. Clear Site Data

Spring Security does not add Clear-Site-Data headers by default. The following Clear-Site-Data header:

Example 227. Clear-Site-Data Example

```
Clear-Site-Data: "cache", "cookies"
```

can be sent on log out with the following configuration:

Example 228. Clear-Site-Data Configuration

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    ServerLogoutHandler securityContext = new
SecurityContextServerLogoutHandler();
    ClearSiteDataServerHttpHeadersWriter writer = new
ClearSiteDataServerHttpHeadersWriter(CACHE, COOKIES);
    ServerLogoutHandler clearSiteData = new
HeaderWriterServerLogoutHandler(writer);
    DelegatingServerLogoutHandler logoutHandler = new
DelegatingServerLogoutHandler(securityContext, clearSiteData);

    http
        // ...
        .logout()
            .logoutHandler(logoutHandler);
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    val securityContext: ServerLogoutHandler =
SecurityContextServerLogoutHandler()
    val writer = ClearSiteDataServerHttpHeadersWriter(CACHE, COOKIES)
    val clearSiteData: ServerLogoutHandler =
HeaderWriterServerLogoutHandler(writer)
    val customLogoutHandler = DelegatingServerLogoutHandler(securityContext,
clearSiteData)

    return http {
        // ...
        logout {
            logoutHandler = customLogoutHandler
        }
    }
}
```

23.3. HTTP

All HTTP based communication should be protected using TLS.

Below you can find details around WebFlux specific features that assist with HTTPS usage.

23.3.1. Redirect to HTTPS

If a client makes a request using HTTP rather than HTTPS, Spring Security can be configured to redirect to HTTPS.

For example, the following Java configuration will redirect any HTTP requests to HTTPS:

Example 229. Redirect to HTTPS

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .redirectToHttps(withDefaults());
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        redirectToHttps { }
    }
}
```

The configuration can easily be wrapped around an if statement to only be turned on in production. Alternatively, it can be enabled by looking for a property about the request that only happens in production. For example, if the production environment adds a header named `X-Forwarded-Proto` the following Java Configuration could be used:

Example 230. Redirect to HTTPS when X-Forwarded

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .redirectToHttps(redirect -> redirect
            .httpsRedirectWhen(e -> e.getRequest().getHeaders().containsKey("X-
Forwarded-Proto"))
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        redirectToHttps {
            httpsRedirectWhen {
                it.request.headers.containsKey("X-Forwarded-Proto")
            }
        }
    }
}
```

23.3.2. Strict Transport Security

Spring Security provides support for Strict Transport Security and enables it by default.

23.3.3. Proxy Server Configuration

Spring Security integrates with proxy servers.

Chapter 24. OAuth2 WebFlux

Spring Security provides OAuth2 and WebFlux integration for reactive applications.

24.1. OAuth 2.0 Login

The OAuth 2.0 Login feature provides an application with the capability to have users log in to the application by using their existing account at an OAuth 2.0 Provider (e.g. GitHub) or OpenID Connect 1.0 Provider (such as Google). OAuth 2.0 Login implements the use cases: "Login with Google" or "Login with GitHub".

OAuth 2.0 Login is implemented by using the **Authorization Code Grant**, as specified in the OAuth 2.0 Authorization Framework and OpenID Connect Core 1.0.

24.1.1. Spring Boot 2.0 Sample

Spring Boot 2.0 brings full auto-configuration capabilities for OAuth 2.0 Login.

This section shows how to configure the **OAuth 2.0 Login WebFlux sample** using *Google* as the *Authentication Provider* and covers the following topics:

- Initial setup
- Setting the redirect URI
- Configure `application.yml`
- Boot up the application

Initial setup

To use Google's OAuth 2.0 authentication system for login, you must set up a project in the Google API Console to obtain OAuth 2.0 credentials.

Google's OAuth 2.0 implementation for authentication conforms to the OpenID Connect 1.0 specification and is OpenID Certified.

Follow the instructions on the OpenID Connect page, starting in the section, "Setting up OAuth 2.0".

After completing the "Obtain OAuth 2.0 credentials" instructions, you should have a new OAuth Client with credentials consisting of a Client ID and a Client Secret.

Setting the redirect URI

The redirect URI is the path in the application that the end-user's user-agent is redirected back to after they have authenticated with Google and have granted access to the OAuth Client *(created in the previous step)* on the Consent page.

In the "Set a redirect URI" sub-section, ensure that the **Authorized redirect URIs** field is set to `http://localhost:8080/login/oauth2/code/google`.

The default redirect URI template is `{baseUrl}/login/oauth2/code/{registrationId}`. The ***registrationId*** is a unique identifier for the ClientRegistration. For our example, the `registrationId` is `google`.

If the OAuth Client is running behind a proxy server, it is recommended to check Proxy Server Configuration to ensure the application is correctly configured. Also, see the supported `URI` template variables for `redirect-uri`.

Configure `application.yml`

Now that you have a new OAuth Client with Google, you need to configure the application to use the OAuth Client for the *authentication flow*. To do so:

1. Go to `application.yml` and set the following configuration:

```
spring:
  security:
    oauth2:
      client:
        registration:   ①
          google:   ②
            client-id: google-client-id
            client-secret: google-client-secret
```

Example 231. OAuth Client properties

① `spring.security.oauth2.client.registration` is the base property prefix for OAuth Client properties.

② Following the base property prefix is the ID for the ClientRegistration, such as google.

2. Replace the values in the `client-id` and `client-secret` property with the OAuth 2.0 credentials you created earlier.

Boot up the application

Launch the Spring Boot 2.0 sample and go to `http://localhost:8080`. You are then redirected to the default *auto-generated* login page, which displays a link for Google.

Click on the Google link, and you are then redirected to Google for authentication.

After authenticating with your Google account credentials, the next page presented to you is the Consent screen. The Consent screen asks you to either allow or deny access to the OAuth Client you created earlier. Click **Allow** to authorize the OAuth Client to access your email address and basic profile information.

At this point, the OAuth Client retrieves your email address and basic profile information from the UserInfo Endpoint and establishes an authenticated session.

24.1.2. Using OpenID Provider Configuration

For well known providers, Spring Security provides the necessary defaults for the OAuth Authorization Provider's configuration. If you are working with your own Authorization Provider that supports OpenID Provider Configuration or Authorization Server Metadata, the OpenID Provider Configuration Response's `issuer-uri` can be used to configure the application.

```
spring:
  security:
    oauth2:
      client:
        provider:
          keycloak:
            issuer-uri: https://idp.example.com/auth/realms/demo
        registration:
          keycloak:
            client-id: spring-security
            client-secret: 6cea952f-10d0-4d00-ac79-cc865820dc2c
```

The `issuer-uri` instructs Spring Security to query in series the endpoints `https://idp.example.com/auth/realms/demo/.well-known/openid-configuration`, `https://idp.example.com/.well-known/openid-configuration/auth/realms/demo`, or `https://idp.example.com/.well-known/oauth-authorization-server/auth/realms/demo` to discover the configuration.

Spring Security will query the endpoints one at a time, stopping at the first that gives a 200 response.

The `client-id` and `client-secret` are linked to the provider because `keycloak` is used for both the provider and the registration.

24.1.3. Explicit OAuth2 Login Configuration

A minimal OAuth2 Login configuration is shown below:

Example 232. Minimal OAuth2 Login

Java

```
@Bean
ReactiveClientRegistrationRepository clientRegistrations() {
    ClientRegistration clientRegistration = ClientRegistrations
            .fromIssuerLocation("https://idp.example.com/auth/realms/demo")
            .clientId("spring-security")
            .clientSecret("6cea952f-10d0-4d00-ac79-cc865820dc2c")
            .build();
    return new InMemoryReactiveClientRegistrationRepository(clientRegistration);
}

@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .oauth2Login(withDefaults());
    return http.build();
}
```

Kotlin

```
@Bean
fun clientRegistrations(): ReactiveClientRegistrationRepository {
    val clientRegistration: ClientRegistration = ClientRegistrations
            .fromIssuerLocation("https://idp.example.com/auth/realms/demo")
            .clientId("spring-security")
            .clientSecret("6cea952f-10d0-4d00-ac79-cc865820dc2c")
            .build()
    return InMemoryReactiveClientRegistrationRepository(clientRegistration)
}

@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        oauth2Login { }
    }
}
```

Additional configuration options can be seen below:

Example 233. Advanced OAuth2 Login

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .oauth2Login(oauth2 -> oauth2
            .authenticationConverter(converter)
            .authenticationManager(manager)
            .authorizedClientRepository(authorizedClients)
            .clientRegistrationRepository(clientRegistrations)
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        oauth2Login {
            authenticationConverter = converter
            authenticationManager = manager
            authorizedClientRepository = authorizedClients
            clientRegistrationRepository = clientRegistration
        }
    }
}
```

You may register a `GrantedAuthoritiesMapper` `@Bean` to have it automatically applied to the default configuration, as shown in the following example:

Example 234. GrantedAuthoritiesMapper Bean

Java

```
@Bean
public GrantedAuthoritiesMapper userAuthoritiesMapper() {
    ...
}

@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .oauth2Login(withDefaults());
    return http.build();
}
```

Kotlin

```
@Bean
fun userAuthoritiesMapper(): GrantedAuthoritiesMapper {
   // ...
}

@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        oauth2Login { }
    }
}
```

24.2. OAuth2 Client

Spring Security's OAuth Support allows obtaining an access token without authenticating. A basic configuration with Spring Boot can be seen below:

```
spring:
  security:
    oauth2:
      client:
        registration:
          github:
            client-id: replace-with-client-id
            client-secret: replace-with-client-secret
            scope: read:user,public_repo
```

You will need to replace the `client-id` and `client-secret` with values registered with GitHub.

The next step is to instruct Spring Security that you wish to act as an OAuth2 Client so that you can obtain an access token.

Example 235. OAuth2 Client

Java

```
@Bean
SecurityWebFilterChain configure(ServerHttpSecurity http) throws Exception {
    http
        // ...
        .oauth2Client(withDefaults());
    return http.build();
}
```

Kotlin

```
@Bean
fun webFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        oauth2Client { }
    }
}
```

You can now leverage Spring Security's WebClient or @RegisteredOAuth2AuthorizedClient support to obtain and use the access token.

24.3. OAuth 2.0 Resource Server

Spring Security supports protecting endpoints using two forms of OAuth 2.0 Bearer Tokens:

- JWT
- Opaque Tokens

This is handy in circumstances where an application has delegated its authority management to an authorization server (for example, Okta or Ping Identity). This authorization server can be consulted by resource servers to authorize requests.

A complete working example for **JWTs** is available in the Spring Security repository.

24.3.1. Minimal Dependencies for JWT

Most Resource Server support is collected into `spring-security-oauth2-resource-server`. However, the support for decoding and verifying JWTs is in `spring-security-oauth2-jose`, meaning that both are necessary in order to have a working resource server that supports JWT-encoded Bearer

Tokens.

24.3.2. Minimal Configuration for JWTs

When using Spring Boot, configuring an application as a resource server consists of two basic steps. First, include the needed dependencies and second, indicate the location of the authorization server.

Specifying the Authorization Server

In a Spring Boot application, to specify which authorization server to use, simply do:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://idp.example.com/issuer
```

Where `https://idp.example.com/issuer` is the value contained in the `iss` claim for JWT tokens that the authorization server will issue. Resource Server will use this property to further self-configure, discover the authorization server's public keys, and subsequently validate incoming JWTs.

To use the `issuer-uri` property, it must also be true that one of `https://idp.example.com/issuer/.well-known/openid-configuration`, `https://idp.example.com/.well-known/openid-configuration/issuer`, or `https://idp.example.com/.well-known/oauth-authorization-server/issuer` is a supported endpoint for the authorization server. This endpoint is referred to as a Provider Configuration endpoint or a Authorization Server Metadata endpoint.

And that's it!

Startup Expectations

When this property and these dependencies are used, Resource Server will automatically configure itself to validate JWT-encoded Bearer Tokens.

It achieves this through a deterministic startup process:

1. Hit the Provider Configuration or Authorization Server Metadata endpoint, processing the response for the `jwks_url` property
2. Configure the validation strategy to query `jwks_url` for valid public keys
3. Configure the validation strategy to validate each JWTs `iss` claim against `https://idp.example.com`.

A consequence of this process is that the authorization server must be up and receiving requests in order for Resource Server to successfully start up.

If the authorization server is down when Resource Server queries it (given appropriate timeouts), then startup will fail.

Runtime Expectations

Once the application is started up, Resource Server will attempt to process any request containing an `Authorization: Bearer` header:

```
GET / HTTP/1.1
Authorization: Bearer some-token-value # Resource Server will process this
```

So long as this scheme is indicated, Resource Server will attempt to process the request according to the Bearer Token specification.

Given a well-formed JWT, Resource Server will:

1. Validate its signature against a public key obtained from the `jwks_url` endpoint during startup and matched against the JWTs header
2. Validate the JWTs `exp` and `nbf` timestamps and the JWTs `iss` claim, and
3. Map each scope to an authority with the prefix `SCOPE_`.

As the authorization server makes available new keys, Spring Security will automatically rotate the keys used to validate the JWT tokens.

The resulting `Authentication#getPrincipal`, by default, is a Spring Security `Jwt` object, and `Authentication#getName` maps to the JWT's `sub` property, if one is present.

From here, consider jumping to:

How to Configure without Tying Resource Server startup to an authorization server's availability

How to Configure without Spring Boot

Specifying the Authorization Server JWK Set Uri Directly

If the authorization server doesn't support any configuration endpoints, or if Resource Server must be able to start up independently from the authorization server, then the `jwk-set-uri` can be supplied as well:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://idp.example.com
          jwk-set-uri: https://idp.example.com/.well-known/jwks.json
```

The JWK Set uri is not standardized, but can typically be found in the authorization server's documentation

Consequently, Resource Server will not ping the authorization server at startup. We still specify the `issuer-uri` so that Resource Server still validates the `iss` claim on incoming JWTs.

This property can also be supplied directly on the DSL.

Overriding or Replacing Boot Auto Configuration

There are two `@Bean` s that Spring Boot generates on Resource Server's behalf.

The first is a `SecurityWebFilterChain` that configures the app as a resource server. When including `spring-security-oauth2-jose`, this `SecurityWebFilterChain` looks like:

Example 236. Resource Server SecurityWebFilterChain

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        .authorizeExchange(exchanges -> exchanges
            .anyExchange().authenticated()
        )
        .oauth2ResourceServer(OAuth2ResourceServerSpec::jwt)
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        authorizeExchange {
            authorize(anyExchange, authenticated)
        }
        oauth2ResourceServer {
            jwt { }
        }
    }
}
```

If the application doesn't expose a `SecurityWebFilterChain` bean, then Spring Boot will expose the above default one.

Replacing this is as simple as exposing the bean within the application:

Example 237. Replacing SecurityWebFilterChain

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        .authorizeExchange(exchanges -> exchanges
            .pathMatchers("/message/**").hasAuthority("SCOPE_message:read")
            .anyExchange().authenticated()
        )
        .oauth2ResourceServer(oauth2 -> oauth2
            .jwt(withDefaults())
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        authorizeExchange {
            authorize("/message/**", hasAuthority("SCOPE_message:read"))
            authorize(anyExchange, authenticated)
        }
        oauth2ResourceServer {
            jwt { }
        }
    }
}
```

The above requires the scope of `message:read` for any URL that starts with `/messages/`.

Methods on the `oauth2ResourceServer` DSL will also override or replace auto configuration.

For example, the second `@Bean` Spring Boot creates is a `ReactiveJwtDecoder`, which decodes `String` tokens into validated instances of `Jwt`:

Example 238. ReactiveJwtDecoder

Java

```
@Bean
public ReactiveJwtDecoder jwtDecoder() {
    return ReactiveJwtDecoders.fromIssuerLocation(issuerUri);
}
```

Kotlin

```
@Bean
fun jwtDecoder(): ReactiveJwtDecoder {
    return ReactiveJwtDecoders.fromIssuerLocation(issuerUri)
}
```

Calling `ReactiveJwtDecoders#fromIssuerLocation` is what invokes the Provider Configuration or Authorization Server Metadata endpoint in order to derive the JWK Set Uri. If the application doesn't expose a `ReactiveJwtDecoder` bean, then Spring Boot will expose the above default one.

And its configuration can be overridden using `jwkSetUri()` or replaced using `decoder()`.

Using `jwkSetUri()`

An authorization server's JWK Set Uri can be configured as a configuration property or it can be supplied in the DSL:

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        .authorizeExchange(exchanges -> exchanges
            .anyExchange().authenticated()
        )
        .oauth2ResourceServer(oauth2 -> oauth2
            .jwt(jwt -> jwt
                .jwkSetUri("https://idp.example.com/.well-known/jwks.json")
            )
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        authorizeExchange {
            authorize(anyExchange, authenticated)
        }
        oauth2ResourceServer {
            jwt {
                jwkSetUri = "https://idp.example.com/.well-known/jwks.json"
            }
        }
    }
}
```

Using `jwkSetUri()` takes precedence over any configuration property.

Using `decoder()`

More powerful than `jwkSetUri()` is `decoder()`, which will completely replace any Boot auto configuration of `JwtDecoder`:

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        .authorizeExchange(exchanges -> exchanges
            .anyExchange().authenticated()
        )
        .oauth2ResourceServer(oauth2 -> oauth2
            .jwt(jwt -> jwt
                .decoder(myCustomDecoder())
            )
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        authorizeExchange {
            authorize(anyExchange, authenticated)
        }
        oauth2ResourceServer {
            jwt {
                jwtDecoder = myCustomDecoder()
            }
        }
    }
}
```

This is handy when deeper configuration, like validation, is necessary.

Exposing a `ReactiveJwtDecoder` `@Bean`

Or, exposing a `ReactiveJwtDecoder` `@Bean` has the same effect as `decoder()`:

Java

```
@Bean
public ReactiveJwtDecoder jwtDecoder() {
    return NimbusReactiveJwtDecoder.withJwkSetUri(jwkSetUri).build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(): ReactiveJwtDecoder {
    return ReactiveJwtDecoders.fromIssuerLocation(issuerUri)
}
```

24.3.3. Configuring Trusted Algorithms

By default, `NimbusReactiveJwtDecoder`, and hence Resource Server, will only trust and verify tokens using `RS256`.

You can customize this via Spring Boot or the NimbusJwtDecoder builder.

Via Spring Boot

The simplest way to set the algorithm is as a property:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jws-algorithm: RS512
          jwk-set-uri: https://idp.example.org/.well-known/jwks.json
```

Using a Builder

For greater power, though, we can use a builder that ships with `NimbusReactiveJwtDecoder`:

Java

```
@Bean
ReactiveJwtDecoder jwtDecoder() {
    return NimbusReactiveJwtDecoder.withJwkSetUri(this.jwkSetUri)
            .jwsAlgorithm(RS512).build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(): ReactiveJwtDecoder {
    return NimbusReactiveJwtDecoder.withJwkSetUri(this.jwkSetUri)
            .jwsAlgorithm(RS512).build()
}
```

Calling `jwsAlgorithm` more than once will configure `NimbusReactiveJwtDecoder` to trust more than one algorithm, like so:

Java

```
@Bean
ReactiveJwtDecoder jwtDecoder() {
    return NimbusReactiveJwtDecoder.withJwkSetUri(this.jwkSetUri)
            .jwsAlgorithm(RS512).jwsAlgorithm(ES512).build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(): ReactiveJwtDecoder {
    return NimbusReactiveJwtDecoder.withJwkSetUri(this.jwkSetUri)
            .jwsAlgorithm(RS512).jwsAlgorithm(ES512).build()
}
```

Or, you can call `jwsAlgorithms`:

Java

```
@Bean
ReactiveJwtDecoder jwtDecoder() {
    return NimbusReactiveJwtDecoder.withJwkSetUri(this.jwkSetUri)
            .jwsAlgorithms(algorithms -> {
                    algorithms.add(RS512);
                    algorithms.add(ES512);
            }).build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(): ReactiveJwtDecoder {
    return NimbusReactiveJwtDecoder.withJwkSetUri(this.jwkSetUri)
            .jwsAlgorithms {
                it.add(RS512)
                it.add(ES512)
            }
            .build()
}
```

Trusting a Single Asymmetric Key

Simpler than backing a Resource Server with a JWK Set endpoint is to hard-code an RSA public key. The public key can be provided via Spring Boot or by Using a Builder.

Via Spring Boot

Specifying a key via Spring Boot is quite simple. The key's location can be specified like so:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          public-key-location: classpath:my-key.pub
```

Or, to allow for a more sophisticated lookup, you can post-process the `RsaKeyConversionServicePostProcessor`:

Example 239. BeanFactoryPostProcessor

Java

```
@Bean
BeanFactoryPostProcessor conversionServiceCustomizer() {
    return beanFactory ->
        beanFactory.getBean(RsaKeyConversionServicePostProcessor.class)
                .setResourceLoader(new CustomResourceLoader());
}
```

Kotlin

```
@Bean
fun conversionServiceCustomizer(): BeanFactoryPostProcessor {
    return BeanFactoryPostProcessor { beanFactory: ConfigurableListableBeanFactory
->
        beanFactory.getBean<RsaKeyConversionServicePostProcessor>()
                .setResourceLoader(CustomResourceLoader())
    }
}
```

Specify your key's location:

```
key.location: hfds://my-key.pub
```

And then autowire the value:

Java

```
@Value("${key.location}")
RSAPublicKey key;
```

Kotlin

```
@Value("\${key.location}")
val key: RSAPublicKey? = null
```

Using a Builder

To wire an `RSAPublicKey` directly, you can simply use the appropriate `NimbusReactiveJwtDecoder` builder, like so:

Java

```
@Bean
public ReactiveJwtDecoder jwtDecoder() {
    return NimbusReactiveJwtDecoder.withPublicKey(this.key).build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(): ReactiveJwtDecoder {
    return NimbusReactiveJwtDecoder.withPublicKey(key).build()
}
```

Trusting a Single Symmetric Key

Using a single symmetric key is also simple. You can simply load in your `SecretKey` and use the appropriate `NimbusReactiveJwtDecoder` builder, like so:

Java

```
@Bean
public ReactiveJwtDecoder jwtDecoder() {
    return NimbusReactiveJwtDecoder.withSecretKey(this.key).build();
}
```

Kotlin

```
@Bean
fun jwtDecoder(): ReactiveJwtDecoder {
    return NimbusReactiveJwtDecoder.withSecretKey(this.key).build()
}
```

Configuring Authorization

A JWT that is issued from an OAuth 2.0 Authorization Server will typically either have a `scope` or `scp` attribute, indicating the scopes (or authorities) it's been granted, for example:

```
{ …, "scope" : "messages contacts"}
```

When this is the case, Resource Server will attempt to coerce these scopes into a list of granted authorities, prefixing each scope with the string "SCOPE_".

This means that to protect an endpoint or method with a scope derived from a JWT, the corresponding expressions should include this prefix:

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        .authorizeExchange(exchanges -> exchanges
            .mvcMatchers("/contacts/**").hasAuthority("SCOPE_contacts")
            .mvcMatchers("/messages/**").hasAuthority("SCOPE_messages")
            .anyExchange().authenticated()
        )
        .oauth2ResourceServer(OAuth2ResourceServerSpec::jwt);
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        authorizeExchange {
            authorize("/contacts/**", hasAuthority("SCOPE_contacts"))
            authorize("/messages/**", hasAuthority("SCOPE_messages"))
            authorize(anyExchange, authenticated)
        }
        oauth2ResourceServer {
            jwt { }
        }
    }
}
```

Or similarly with method security:

Java

```
@PreAuthorize("hasAuthority('SCOPE_messages')")
public Flux<Message> getMessages(...) {}
```

Kotlin

```
@PreAuthorize("hasAuthority('SCOPE_messages')")
fun getMessages(): Flux<Message> { }
```

Extracting Authorities Manually

However, there are a number of circumstances where this default is insufficient. For example,

some authorization servers don't use the `scope` attribute, but instead have their own custom attribute. Or, at other times, the resource server may need to adapt the attribute or a composition of attributes into internalized authorities.

To this end, the DSL exposes `jwtAuthenticationConverter()`:

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        .authorizeExchange(exchanges -> exchanges
            .anyExchange().authenticated()
        )
        .oauth2ResourceServer(oauth2 -> oauth2
            .jwt(jwt -> jwt
                .jwtAuthenticationConverter(grantedAuthoritiesExtractor())
            )
        );
    return http.build();
}

Converter<Jwt, Mono<AbstractAuthenticationToken>> grantedAuthoritiesExtractor() {
    JwtAuthenticationConverter jwtAuthenticationConverter =
            new JwtAuthenticationConverter();
    jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter
            (new GrantedAuthoritiesExtractor());
    return new
ReactiveJwtAuthenticationConverterAdapter(jwtAuthenticationConverter);
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        authorizeExchange {
            authorize(anyExchange, authenticated)
        }
        oauth2ResourceServer {
            jwt {
                jwtAuthenticationConverter = grantedAuthoritiesExtractor()
            }
        }
    }
}

fun grantedAuthoritiesExtractor(): Converter<Jwt,
Mono<AbstractAuthenticationToken>> {
    val jwtAuthenticationConverter = JwtAuthenticationConverter()

jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(GrantedAuthoritiesExt
ractor())
    return ReactiveJwtAuthenticationConverterAdapter(jwtAuthenticationConverter)
}
```

which is responsible for converting a `Jwt` into an `Authentication`. As part of its configuration, we can supply a subsidiary converter to go from `Jwt` to a `Collection` of granted authorities.

That final converter might be something like `GrantedAuthoritiesExtractor` below:

Java

```
static class GrantedAuthoritiesExtractor
        implements Converter<Jwt, Collection<GrantedAuthority>> {

    public Collection<GrantedAuthority> convert(Jwt jwt) {
        Collection<?> authorities = (Collection<?>)
                jwt.getClaims().getOrDefault("mycustomclaim",
Collections.emptyList());

        return authorities.stream()
                .map(Object::toString)
                .map(SimpleGrantedAuthority::new)
                .collect(Collectors.toList());
    }
}
```

Kotlin

```
internal class GrantedAuthoritiesExtractor : Converter<Jwt,
Collection<GrantedAuthority>> {
    override fun convert(jwt: Jwt): Collection<GrantedAuthority> {
        val authorities: List<Any> = jwt.claims
                .getOrDefault("mycustomclaim", emptyList<Any>()) as List<Any>
        return authorities
                .map { it.toString() }
                .map { SimpleGrantedAuthority(it) }
    }
}
```

For more flexibility, the DSL supports entirely replacing the converter with any class that implements `Converter<Jwt, Mono<AbstractAuthenticationToken>>`:

Java

```
static class CustomAuthenticationConverter implements Converter<Jwt,
Mono<AbstractAuthenticationToken>> {
    public AbstractAuthenticationToken convert(Jwt jwt) {
        return Mono.just(jwt).map(this::doConversion);
    }
}
```

Kotlin

```
internal class CustomAuthenticationConverter : Converter<Jwt,
Mono<AbstractAuthenticationToken>> {
    override fun convert(jwt: Jwt): Mono<AbstractAuthenticationToken> {
        return Mono.just(jwt).map(this::doConversion)
    }
}
```

Configuring Validation

Using minimal Spring Boot configuration, indicating the authorization server's issuer uri, Resource Server will default to verifying the `iss` claim as well as the `exp` and `nbf` timestamp claims.

In circumstances where validation needs to be customized, Resource Server ships with two standard validators and also accepts custom `OAuth2TokenValidator` instances.

Customizing Timestamp Validation

JWT's typically have a window of validity, with the start of the window indicated in the `nbf` claim and the end indicated in the `exp` claim.

However, every server can experience clock drift, which can cause tokens to appear expired to one server, but not to another. This can cause some implementation heartburn as the number of collaborating servers increases in a distributed system.

Resource Server uses `JwtTimestampValidator` to verify a token's validity window, and it can be configured with a `clockSkew` to alleviate the above problem:

Java

```
@Bean
ReactiveJwtDecoder jwtDecoder() {
     NimbusReactiveJwtDecoder jwtDecoder = (NimbusReactiveJwtDecoder)
             ReactiveJwtDecoders.fromIssuerLocation(issuerUri);

     OAuth2TokenValidator<Jwt> withClockSkew = new
DelegatingOAuth2TokenValidator<>(
             new JwtTimestampValidator(Duration.ofSeconds(60)),
             new IssuerValidator(issuerUri));

     jwtDecoder.setJwtValidator(withClockSkew);

     return jwtDecoder;
}
```

Kotlin

```
@Bean
fun jwtDecoder(): ReactiveJwtDecoder {
    val jwtDecoder = ReactiveJwtDecoders.fromIssuerLocation(issuerUri) as
NimbusReactiveJwtDecoder
    val withClockSkew: OAuth2TokenValidator<Jwt> = DelegatingOAuth2TokenValidator(
            JwtTimestampValidator(Duration.ofSeconds(60)),
            JwtIssuerValidator(issuerUri))
    jwtDecoder.setJwtValidator(withClockSkew)
    return jwtDecoder
}
```

By default, Resource Server configures a clock skew of 60 seconds.

Configuring a Custom Validator

Adding a check for the `aud` claim is simple with the `OAuth2TokenValidator` API:

Java

```
public class AudienceValidator implements OAuth2TokenValidator<Jwt> {
    OAuth2Error error = new OAuth2Error("invalid_token", "The required audience is
missing", null);

    public OAuth2TokenValidatorResult validate(Jwt jwt) {
        if (jwt.getAudience().contains("messaging")) {
            return OAuth2TokenValidatorResult.success();
        } else {
            return OAuth2TokenValidatorResult.failure(error);
        }
    }
}
```

Kotlin

```
class AudienceValidator : OAuth2TokenValidator<Jwt> {
    var error: OAuth2Error = OAuth2Error("invalid_token", "The required audience
is missing", null)
    override fun validate(jwt: Jwt): OAuth2TokenValidatorResult {
        return if (jwt.audience.contains("messaging")) {
            OAuth2TokenValidatorResult.success()
        } else {
            OAuth2TokenValidatorResult.failure(error)
        }
    }
}
```

Then, to add into a resource server, it's a matter of specifying the `ReactiveJwtDecoder` instance:

Java

```
@Bean
ReactiveJwtDecoder jwtDecoder() {
    NimbusReactiveJwtDecoder jwtDecoder = (NimbusReactiveJwtDecoder)
            ReactiveJwtDecoders.fromIssuerLocation(issuerUri);

    OAuth2TokenValidator<Jwt> audienceValidator = new AudienceValidator();
    OAuth2TokenValidator<Jwt> withIssuer =
JwtValidators.createDefaultWithIssuer(issuerUri);
    OAuth2TokenValidator<Jwt> withAudience = new
DelegatingOAuth2TokenValidator<>(withIssuer, audienceValidator);

    jwtDecoder.setJwtValidator(withAudience);

    return jwtDecoder;
}
```

Kotlin

```
@Bean
fun jwtDecoder(): ReactiveJwtDecoder {
    val jwtDecoder = ReactiveJwtDecoders.fromIssuerLocation(issuerUri) as
NimbusReactiveJwtDecoder
    val audienceValidator: OAuth2TokenValidator<Jwt> = AudienceValidator()
    val withIssuer: OAuth2TokenValidator<Jwt> =
JwtValidators.createDefaultWithIssuer(issuerUri)
    val withAudience: OAuth2TokenValidator<Jwt> =
DelegatingOAuth2TokenValidator(withIssuer, audienceValidator)
    jwtDecoder.setJwtValidator(withAudience)
    return jwtDecoder
}
```

Minimal Dependencies for Introspection

As described in Minimal Dependencies for JWT most of Resource Server support is collected in `spring-security-oauth2-resource-server`. However unless a custom `ReactiveOpaqueTokenIntrospector` is provided, the Resource Server will fallback to ReactiveOpaqueTokenIntrospector. Meaning that both `spring-security-oauth2-resource-server` and `oauth2-oidc-sdk` are necessary in order to have a working minimal Resource Server that supports opaque Bearer Tokens. Please refer to `spring-security-oauth2-resource-server` in order to determin the correct version for `oauth2-oidc-sdk`.

Minimal Configuration for Introspection

Typically, an opaque token can be verified via an OAuth 2.0 Introspection Endpoint, hosted by the authorization server. This can be handy when revocation is a requirement.

When using Spring Boot, configuring an application as a resource server that uses introspection

consists of two basic steps. First, include the needed dependencies and second, indicate the introspection endpoint details.

Specifying the Authorization Server

To specify where the introspection endpoint is, simply do:

```
security:
  oauth2:
    resourceserver:
      opaque-token:
        introspection-uri: https://idp.example.com/introspect
        client-id: client
        client-secret: secret
```

Where `https://idp.example.com/introspect` is the introspection endpoint hosted by your authorization server and `client-id` and `client-secret` are the credentials needed to hit that endpoint.

Resource Server will use these properties to further self-configure and subsequently validate incoming JWTs.

When using introspection, the authorization server's word is the law. If the authorization server responses that the token is valid, then it is.

And that's it!

Startup Expectations

When this property and these dependencies are used, Resource Server will automatically configure itself to validate Opaque Bearer Tokens.

This startup process is quite a bit simpler than for JWTs since no endpoints need to be discovered and no additional validation rules get added.

Runtime Expectations

Once the application is started up, Resource Server will attempt to process any request containing an `Authorization: Bearer` header:

```
GET / HTTP/1.1
Authorization: Bearer some-token-value # Resource Server will process this
```

So long as this scheme is indicated, Resource Server will attempt to process the request according to the Bearer Token specification.

Given an Opaque Token, Resource Server will

1. Query the provided introspection endpoint using the provided credentials and the token

2. Inspect the response for an `{ 'active' : true }` attribute
3. Map each scope to an authority with the prefix `SCOPE_`

The resulting `Authentication#getPrincipal`, by default, is a Spring Security `OAuth2AuthenticatedPrincipal` object, and `Authentication#getName` maps to the token's `sub` property, if one is present.

From here, you may want to jump to:

- Looking Up Attributes Post-Authentication
- Extracting Authorities Manually
- Using Introspection with JWTs

Looking Up Attributes Post-Authentication

Once a token is authenticated, an instance of `BearerTokenAuthentication` is set in the `SecurityContext`.

This means that it's available in `@Controller` methods when using `@EnableWebFlux` in your configuration:

Java

```
@GetMapping("/foo")
public Mono<String> foo(BearerTokenAuthentication authentication) {
    return Mono.just(authentication.getTokenAttributes().get("sub") + " is the
subject");
}
```

Kotlin

```
@GetMapping("/foo")
fun foo(authentication: BearerTokenAuthentication): Mono<String> {
    return Mono.just(authentication.tokenAttributes["sub"].toString() + " is the
subject")
}
```

Since `BearerTokenAuthentication` holds an `OAuth2AuthenticatedPrincipal`, that also means that it's available to controller methods, too:

Java

```
@GetMapping("/foo")
public Mono<String> foo(@AuthenticationPrincipal OAuth2AuthenticatedPrincipal
principal) {
    return Mono.just(principal.getAttribute("sub") + " is the subject");
}
```

Kotlin

```
@GetMapping("/foo")
fun foo(@AuthenticationPrincipal principal: OAuth2AuthenticatedPrincipal):
Mono<String> {
    return Mono.just(principal.getAttribute<Any>("sub").toString() + " is the
subject")
}
```

Looking Up Attributes Via SpEL

Of course, this also means that attributes can be accessed via SpEL.

For example, if using `@EnableReactiveMethodSecurity` so that you can use `@PreAuthorize` annotations, you can do:

Java

```
@PreAuthorize("principal?.attributes['sub'] == 'foo'")
public Mono<String> forFoosEyesOnly() {
    return Mono.just("foo");
}
```

Kotlin

```
@PreAuthorize("principal.attributes['sub'] == 'foo'")
fun forFoosEyesOnly(): Mono<String> {
    return Mono.just("foo")
}
```

Overriding or Replacing Boot Auto Configuration

There are two `@Bean` s that Spring Boot generates on Resource Server's behalf.

The first is a `SecurityWebFilterChain` that configures the app as a resource server. When use Opaque Token, this `SecurityWebFilterChain` looks like:

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        .authorizeExchange(exchanges -> exchanges
            .anyExchange().authenticated()
        )

.oauth2ResourceServer(ServerHttpSecurity.OAuth2ResourceServerSpec::opaqueToken)
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        authorizeExchange {
            authorize(anyExchange, authenticated)
        }
        oauth2ResourceServer {
            opaqueToken { }
        }
    }
}
```

If the application doesn't expose a `SecurityWebFilterChain` bean, then Spring Boot will expose the above default one.

Replacing this is as simple as exposing the bean within the application:

Example 240. Replacing SecurityWebFilterChain

Java

```
@EnableWebFluxSecurity
public class MyCustomSecurityConfiguration {
    @Bean
    SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http
            .authorizeExchange(exchanges -> exchanges
                .pathMatchers("/messages/**").hasAuthority("SCOPE_message:read")
                .anyExchange().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .opaqueToken(opaqueToken -> opaqueToken
                    .introspector(myIntrospector())
                )
            );
        return http.build();
    }
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        authorizeExchange {
            authorize("/messages/**", hasAuthority("SCOPE_message:read"))
            authorize(anyExchange, authenticated)
        }
        oauth2ResourceServer {
            opaqueToken {
                introspector = myIntrospector()
            }
        }
    }
}
```

The above requires the scope of `message:read` for any URL that starts with `/messages/`.

Methods on the `oauth2ResourceServer` DSL will also override or replace auto configuration.

For example, the second `@Bean` Spring Boot creates is a `ReactiveOpaqueTokenIntrospector`, which decodes `String` tokens into validated instances of `OAuth2AuthenticatedPrincipal`:

Java

```
@Bean
public ReactiveOpaqueTokenIntrospector introspector() {
    return new NimbusReactiveOpaqueTokenIntrospector(introspectionUri, clientId,
clientSecret);
}
```

Kotlin

```
@Bean
fun introspector(): ReactiveOpaqueTokenIntrospector {
    return NimbusReactiveOpaqueTokenIntrospector(introspectionUri, clientId,
clientSecret)
}
```

If the application doesn't expose a `ReactiveOpaqueTokenIntrospector` bean, then Spring Boot will expose the above default one.

And its configuration can be overridden using `introspectionUri()` and `introspectionClientCredentials()` or replaced using `introspector()`.

Using `introspectionUri()`

An authorization server's Introspection Uri can be configured as a configuration property or it can be supplied in the DSL:

Java

```
@EnableWebFluxSecurity
public class DirectlyConfiguredIntrospectionUri {
    @Bean
    SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http
            .authorizeExchange(exchanges -> exchanges
                .anyExchange().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .opaqueToken(opaqueToken -> opaqueToken
                    .introspectionUri("https://idp.example.com/introspect")
                    .introspectionClientCredentials("client", "secret")
                )
            );
        return http.build();
    }
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        authorizeExchange {
            authorize(anyExchange, authenticated)
        }
        oauth2ResourceServer {
            opaqueToken {
                introspectionUri = "https://idp.example.com/introspect"
                introspectionClientCredentials("client", "secret")
            }
        }
    }
}
```

Using `introspectionUri()` takes precedence over any configuration property.

Using `introspector()`

More powerful than `introspectionUri()` is `introspector()`, which will completely replace any Boot auto configuration of `ReactiveOpaqueTokenIntrospector`:

Java

```
@EnableWebFluxSecurity
public class DirectlyConfiguredIntrospector {
    @Bean
    SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http
            .authorizeExchange(exchanges -> exchanges
                .anyExchange().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .opaqueToken(opaqueToken -> opaqueToken
                    .introspector(myCustomIntrospector())
                )
            );
        return http.build();
    }
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        authorizeExchange {
            authorize(anyExchange, authenticated)
        }
        oauth2ResourceServer {
            opaqueToken {
                introspector = myCustomIntrospector()
            }
        }
    }
}
```

This is handy when deeper configuration, like authority mappingor JWT revocation is necessary.

Exposing a `ReactiveOpaqueTokenIntrospector` `@Bean`

Or, exposing a `ReactiveOpaqueTokenIntrospector` `@Bean` has the same effect as `introspector()`:

Java

```
@Bean
public ReactiveOpaqueTokenIntrospector introspector() {
    return new NimbusReactiveOpaqueTokenIntrospector(introspectionUri, clientId,
clientSecret);
}
```

Kotlin

```
@Bean
fun introspector(): ReactiveOpaqueTokenIntrospector {
    return NimbusReactiveOpaqueTokenIntrospector(introspectionUri, clientId,
clientSecret)
}
```

Configuring Authorization

An OAuth 2.0 Introspection endpoint will typically return a `scope` attribute, indicating the scopes (or authorities) it's been granted, for example:

```
{ …, "scope" : "messages contacts"}
```

When this is the case, Resource Server will attempt to coerce these scopes into a list of granted authorities, prefixing each scope with the string "SCOPE_".

This means that to protect an endpoint or method with a scope derived from an Opaque Token, the corresponding expressions should include this prefix:

Java

```
@EnableWebFluxSecurity
public class MappedAuthorities {
    @Bean
    SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http
            .authorizeExchange(exchange -> exchange
                .pathMatchers("/contacts/**").hasAuthority("SCOPE_contacts")
                .pathMatchers("/messages/**").hasAuthority("SCOPE_messages")
                .anyExchange().authenticated()
            )

.oauth2ResourceServer(ServerHttpSecurity.OAuth2ResourceServerSpec::opaqueToken);
        return http.build();
    }
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        authorizeExchange {
            authorize("/contacts/**", hasAuthority("SCOPE_contacts"))
            authorize("/messages/**", hasAuthority("SCOPE_messages"))
            authorize(anyExchange, authenticated)
        }
        oauth2ResourceServer {
            opaqueToken { }
        }
    }
}
```

Or similarly with method security:

Java

```
@PreAuthorize("hasAuthority('SCOPE_messages')")
public Flux<Message> getMessages(...) {}
```

Kotlin

```
@PreAuthorize("hasAuthority('SCOPE_messages')")
fun getMessages(): Flux<Message> { }
```

Extracting Authorities Manually

By default, Opaque Token support will extract the scope claim from an introspection response and parse it into individual `GrantedAuthority` instances.

For example, if the introspection response were:

```
{
    "active" : true,
    "scope" : "message:read message:write"
}
```

Then Resource Server would generate an `Authentication` with two authorities, one for `message:read` and the other for `message:write`.

This can, of course, be customized using a custom `ReactiveOpaqueTokenIntrospector` that takes a look at the attribute set and converts in its own way:

Java

```
public class CustomAuthoritiesOpaqueTokenIntrospector implements
ReactiveOpaqueTokenIntrospector {
    private ReactiveOpaqueTokenIntrospector delegate =
            new
NimbusReactiveOpaqueTokenIntrospector("https://idp.example.org/introspect",
"client", "secret");

    public Mono<OAuth2AuthenticatedPrincipal> introspect(String token) {
        return this.delegate.introspect(token)
                .map(principal -> new DefaultOAuth2AuthenticatedPrincipal(
                        principal.getName(), principal.getAttributes(),
extractAuthorities(principal)));
    }

    private Collection<GrantedAuthority>
extractAuthorities(OAuth2AuthenticatedPrincipal principal) {
        List<String> scopes =
principal.getAttribute(OAuth2IntrospectionClaimNames.SCOPE);
        return scopes.stream()
                .map(SimpleGrantedAuthority::new)
                .collect(Collectors.toList());
    }
}
```

Kotlin

```
class CustomAuthoritiesOpaqueTokenIntrospector : ReactiveOpaqueTokenIntrospector {
    private val delegate: ReactiveOpaqueTokenIntrospector =
NimbusReactiveOpaqueTokenIntrospector("https://idp.example.org/introspect",
"client", "secret")
    override fun introspect(token: String): Mono<OAuth2AuthenticatedPrincipal> {
        return delegate.introspect(token)
                .map { principal: OAuth2AuthenticatedPrincipal ->
                    DefaultOAuth2AuthenticatedPrincipal(
                            principal.name, principal.attributes,
extractAuthorities(principal))
                }
    }

    private fun extractAuthorities(principal: OAuth2AuthenticatedPrincipal):
Collection<GrantedAuthority> {
        val scopes =
principal.getAttribute<List<String>>(OAuth2IntrospectionClaimNames.SCOPE)
        return scopes
                .map { SimpleGrantedAuthority(it) }
    }
}
```

Thereafter, this custom introspector can be configured simply by exposing it as a `@Bean`:

Java

```
@Bean
public ReactiveOpaqueTokenIntrospector introspector() {
    return new CustomAuthoritiesOpaqueTokenIntrospector();
}
```

Kotlin

```
@Bean
fun introspector(): ReactiveOpaqueTokenIntrospector {
    return CustomAuthoritiesOpaqueTokenIntrospector()
}
```

Using Introspection with JWTs

A common question is whether or not introspection is compatible with JWTs. Spring Security's Opaque Token support has been designed to not care about the format of the token — it will gladly pass any token to the introspection endpoint provided.

So, let's say that you've got a requirement that requires you to check with the authorization server on each request, in case the JWT has been revoked.

Even though you are using the JWT format for the token, your validation method is introspection, meaning you'd want to do:

```
spring:
  security:
    oauth2:
      resourceserver:
        opaque-token:
          introspection-uri: https://idp.example.org/introspection
          client-id: client
          client-secret: secret
```

In this case, the resulting `Authentication` would be `BearerTokenAuthentication`. Any attributes in the corresponding `OAuth2AuthenticatedPrincipal` would be whatever was returned by the introspection endpoint.

But, let's say that, oddly enough, the introspection endpoint only returns whether or not the token is active. Now what?

In this case, you can create a custom `ReactiveOpaqueTokenIntrospector` that still hits the endpoint, but then updates the returned principal to have the JWTs claims as the attributes:

Java

```
public class JwtOpaqueTokenIntrospector implements ReactiveOpaqueTokenIntrospector
{
    private ReactiveOpaqueTokenIntrospector delegate =
            new
NimbusReactiveOpaqueTokenIntrospector("https://idp.example.org/introspect",
"client", "secret");
    private ReactiveJwtDecoder jwtDecoder = new NimbusReactiveJwtDecoder(new
ParseOnlyJWTProcessor());

    public Mono<OAuth2AuthenticatedPrincipal> introspect(String token) {
        return this.delegate.introspect(token)
                .flatMap(principal -> this.jwtDecoder.decode(token))
                .map(jwt -> new
DefaultOAuth2AuthenticatedPrincipal(jwt.getClaims(), NO_AUTHORITIES));
    }

    private static class ParseOnlyJWTProcessor implements Converter<JWT,
Mono<JWTClaimsSet>> {
        public Mono<JWTClaimsSet> convert(JWT jwt) {
            try {
                return Mono.just(jwt.getJWTClaimsSet());
            } catch (Exception ex) {
                return Mono.error(ex);
            }
        }
    }
}
```

Kotlin

```
class JwtOpaqueTokenIntrospector : ReactiveOpaqueTokenIntrospector {
    private val delegate: ReactiveOpaqueTokenIntrospector =
NimbusReactiveOpaqueTokenIntrospector("https://idp.example.org/introspect",
"client", "secret")
    private val jwtDecoder: ReactiveJwtDecoder =
NimbusReactiveJwtDecoder(ParseOnlyJWTProcessor())
    override fun introspect(token: String): Mono<OAuth2AuthenticatedPrincipal> {
        return delegate.introspect(token)
                .flatMap { jwtDecoder.decode(token) }
                .map { jwt: Jwt -> DefaultOAuth2AuthenticatedPrincipal(jwt.claims,
NO_AUTHORITIES) }
    }

    private class ParseOnlyJWTProcessor : Converter<JWT, Mono<JWTClaimsSet>> {
        override fun convert(jwt: JWT): Mono<JWTClaimsSet> {
            return try {
                Mono.just(jwt.jwtClaimsSet)
            } catch (e: Exception) {
                Mono.error(e)
            }
        }
    }
}
```

Thereafter, this custom introspector can be configured simply by exposing it as a `@Bean`:

Java

```
@Bean
public ReactiveOpaqueTokenIntrospector introspector() {
    return new JwtOpaqueTokenIntropsector();
}
```

Kotlin

```
@Bean
fun introspector(): ReactiveOpaqueTokenIntrospector {
    return JwtOpaqueTokenIntrospector()
}
```

Calling a `/userinfo` Endpoint

Generally speaking, a Resource Server doesn't care about the underlying user, but instead about the authorities that have been granted.

That said, at times it can be valuable to tie the authorization statement back to a user.

If an application is also using `spring-security-oauth2-client`, having set up the appropriate `ClientRegistrationRepository`, then this is quite simple with a custom `OpaqueTokenIntrospector`. This implementation below does three things:

- Delegates to the introspection endpoint, to affirm the token's validity
- Looks up the appropriate client registration associated with the `/userinfo` endpoint
- Invokes and returns the response from the `/userinfo` endpoint

Java

```
public class UserInfoOpaqueTokenIntrospector implements
ReactiveOpaqueTokenIntrospector {
    private final ReactiveOpaqueTokenIntrospector delegate =
            new
NimbusReactiveOpaqueTokenIntrospector("https://idp.example.org/introspect",
"client", "secret");
    private final ReactiveOAuth2UserService<OAuth2UserRequest, OAuth2User>
oauth2UserService =
            new DefaultReactiveOAuth2UserService();

    private final ReactiveClientRegistrationRepository repository;

    // ... constructor

    @Override
    public Mono<OAuth2AuthenticatedPrincipal> introspect(String token) {
        return Mono.zip(this.delegate.introspect(token),
this.repository.findByRegistrationId("registration-id"))
                .map(t -> {
                    OAuth2AuthenticatedPrincipal authorized = t.getT1();
                    ClientRegistration clientRegistration = t.getT2();
                    Instant issuedAt = authorized.getAttribute(ISSUED_AT);
                    Instant expiresAt =
authorized.getAttribute(OAuth2IntrospectionClaimNames.EXPIRES_AT);
                    OAuth2AccessToken accessToken = new OAuth2AccessToken(BEARER,
token, issuedAt, expiresAt);
                    return new OAuth2UserRequest(clientRegistration, accessToken);
                })
                .flatMap(this.oauth2UserService::loadUser);
    }
}
```

Kotlin

```
class UserInfoOpaqueTokenIntrospector : ReactiveOpaqueTokenIntrospector {
    private val delegate: ReactiveOpaqueTokenIntrospector =
NimbusReactiveOpaqueTokenIntrospector("https://idp.example.org/introspect",
"client", "secret")
    private val oauth2UserService: ReactiveOAuth2UserService<OAuth2UserRequest,
OAuth2User> = DefaultReactiveOAuth2UserService()
    private val repository: ReactiveClientRegistrationRepository? = null

    // ... constructor
    override fun introspect(token: String?): Mono<OAuth2AuthenticatedPrincipal> {
        return Mono.zip<OAuth2AuthenticatedPrincipal,
ClientRegistration>(delegate.introspect(token),
repository!!.findByRegistrationId("registration-id"))
                .map<OAuth2UserRequest> { t: Tuple2<OAuth2AuthenticatedPrincipal,
ClientRegistration> ->
                    val authorized = t.t1
                    val clientRegistration = t.t2
                    val issuedAt: Instant? = authorized.getAttribute(ISSUED_AT)
                    val expiresAt: Instant? =
authorized.getAttribute(OAuth2IntrospectionClaimNames.EXPIRES_AT)
                    val accessToken = OAuth2AccessToken(BEARER, token, issuedAt,
expiresAt)
                    OAuth2UserRequest(clientRegistration, accessToken)
                }
                .flatMap { userRequest: OAuth2UserRequest ->
oauth2UserService.loadUser(userRequest) }
    }
}
```

If you aren't using `spring-security-oauth2-client`, it's still quite simple. You will simply need to invoke the `/userinfo` with your own instance of `WebClient`:

Java

```
public class UserInfoOpaqueTokenIntrospector implements
ReactiveOpaqueTokenIntrospector {
    private final ReactiveOpaqueTokenIntrospector delegate =
            new
NimbusReactiveOpaqueTokenIntrospector("https://idp.example.org/introspect",
"client", "secret");
    private final WebClient rest = WebClient.create();

    @Override
    public Mono<OAuth2AuthenticatedPrincipal> introspect(String token) {
        return this.delegate.introspect(token)
                .map(this::makeUserInfoRequest);
    }
}
```

Kotlin

```
class UserInfoOpaqueTokenIntrospector : ReactiveOpaqueTokenIntrospector {
    private val delegate: ReactiveOpaqueTokenIntrospector =
NimbusReactiveOpaqueTokenIntrospector("https://idp.example.org/introspect",
"client", "secret")
    private val rest: WebClient = WebClient.create()

    override fun introspect(token: String): Mono<OAuth2AuthenticatedPrincipal> {
        return delegate.introspect(token)
                .map(this::makeUserInfoRequest)
    }
}
```

Either way, having created your `ReactiveOpaqueTokenIntrospector`, you should publish it as a `@Bean` to override the defaults:

Java

```
@Bean
ReactiveOpaqueTokenIntrospector introspector() {
    return new UserInfoOpaqueTokenIntrospector();
}
```

Kotlin

```
@Bean
fun introspector(): ReactiveOpaqueTokenIntrospector {
    return UserInfoOpaqueTokenIntrospector()
}
```

24.3.4. Multi-tenancy

A resource server is considered multi-tenant when there are multiple strategies for verifying a bearer token, keyed by some tenant identifier.

For example, your resource server may accept bearer tokens from two different authorization servers. Or, your authorization server may represent a multiplicity of issuers.

In each case, there are two things that need to be done and trade-offs associated with how you choose to do them:

1. Resolve the tenant
2. Propagate the tenant

Resolving the Tenant By Claim

One way to differentiate tenants is by the issuer claim. Since the issuer claim accompanies signed JWTs, this can be done with the `JwtIssuerReactiveAuthenticationManagerResolver`, like so:

Java

```
JwtIssuerReactiveAuthenticationManagerResolver authenticationManagerResolver = new
JwtIssuerReactiveAuthenticationManagerResolver
    ("https://idp.example.org/issuerOne", "https://idp.example.org/issuerTwo");

http
    .authorizeExchange(exchanges -> exchanges
        .anyExchange().authenticated()
    )
    .oauth2ResourceServer(oauth2 -> oauth2
        .authenticationManagerResolver(authenticationManagerResolver)
    );
```

Kotlin

```
val customAuthenticationManagerResolver =
JwtIssuerReactiveAuthenticationManagerResolver("https://idp.example.org/issuerOne"
, "https://idp.example.org/issuerTwo")

return http {
    authorizeExchange {
        authorize(anyExchange, authenticated)
    }
    oauth2ResourceServer {
        authenticationManagerResolver = customAuthenticationManagerResolver
    }
}
```

This is nice because the issuer endpoints are loaded lazily. In fact, the corresponding `JwtReactiveAuthenticationManager` is instantiated only when the first request with the corresponding issuer is sent. This allows for an application startup that is independent from those authorization servers being up and available.

Dynamic Tenants

Of course, you may not want to restart the application each time a new tenant is added. In this case, you can configure the `JwtIssuerReactiveAuthenticationManagerResolver` with a repository of `ReactiveAuthenticationManager` instances, which you can edit at runtime, like so:

Java

```
private Mono<ReactiveAuthenticationManager> addManager(
        Map<String, ReactiveAuthenticationManager> authenticationManagers, String
issuer) {

    return Mono.fromCallable(() -> ReactiveJwtDecoders.fromIssuerLocation(issuer))
            .subscribeOn(Schedulers.boundedElastic())
            .map(JwtReactiveAuthenticationManager::new)
            .doOnNext(authenticationManager -> authenticationManagers.put(issuer,
authenticationManager));
}

// ...

JwtIssuerReactiveAuthenticationManagerResolver authenticationManagerResolver =
        new
JwtIssuerReactiveAuthenticationManagerResolver(authenticationManagers::get);

http
    .authorizeExchange(exchanges -> exchanges
        .anyExchange().authenticated()
    )
    .oauth2ResourceServer(oauth2 -> oauth2
        .authenticationManagerResolver(authenticationManagerResolver)
    );
```

Kotlin

```
private fun addManager(
        authenticationManagers: MutableMap<String, ReactiveAuthenticationManager>,
issuer: String): Mono<JwtReactiveAuthenticationManager> {
    return Mono.fromCallable { ReactiveJwtDecoders.fromIssuerLocation(issuer) }
            .subscribeOn(Schedulers.boundedElastic())
            .map { jwtDecoder: ReactiveJwtDecoder ->
JwtReactiveAuthenticationManager(jwtDecoder) }
            .doOnNext { authenticationManager: JwtReactiveAuthenticationManager ->
authenticationManagers[issuer] = authenticationManager }
}

// ...

var customAuthenticationManagerResolver =
JwtIssuerReactiveAuthenticationManagerResolver(authenticationManagers::get)
return http {
    authorizeExchange {
        authorize(anyExchange, authenticated)
    }
    oauth2ResourceServer {
        authenticationManagerResolver = customAuthenticationManagerResolver
    }
}
```

In this case, you construct `JwtIssuerReactiveAuthenticationManagerResolver` with a strategy for obtaining the `ReactiveAuthenticationManager` given the issuer. This approach allows us to add and remove elements from the repository (shown as a `Map` in the snippet) at runtime.

It would be unsafe to simply take any issuer and construct an `ReactiveAuthenticationManager` from it. The issuer should be one that the code can verify from a trusted source like an allowed list of issuers.

24.3.5. Bearer Token Resolution

By default, Resource Server looks for a bearer token in the `Authorization` header. This, however, can be customized.

For example, you may have a need to read the bearer token from a custom header. To achieve this, you can wire an instance of `ServerBearerTokenAuthenticationConverter` into the DSL, as you can see in the following example:

Example 241. Custom Bearer Token Header

Java

```
ServerBearerTokenAuthenticationConverter converter = new
ServerBearerTokenAuthenticationConverter();
converter.setBearerTokenHeaderName(HttpHeaders.PROXY_AUTHORIZATION);
http
    .oauth2ResourceServer(oauth2 -> oauth2
        .bearerTokenConverter(converter)
    );
```

Kotlin

```
val converter = ServerBearerTokenAuthenticationConverter()
converter.setBearerTokenHeaderName(HttpHeaders.PROXY_AUTHORIZATION)
return http {
    oauth2ResourceServer {
        bearerTokenConverter = converter
    }
}
```

24.3.6. Bearer Token Propagation

Now that you're in possession of a bearer token, it might be handy to pass that to downstream services. This is quite simple with `ServerBearerExchangeFilterFunction`, which you can see in the following example:

Java

```
@Bean
public WebClient rest() {
    return WebClient.builder()
            .filter(new ServerBearerExchangeFilterFunction())
            .build();
}
```

Kotlin

```
@Bean
fun rest(): WebClient {
    return WebClient.builder()
            .filter(ServerBearerExchangeFilterFunction())
            .build()
}
```

When the above `WebClient` is used to perform requests, Spring Security will look up the current `Authentication` and extract any `AbstractOAuth2Token` credential. Then, it will propagate that token in the `Authorization` header.

For example:

Java

```
this.rest.get()
        .uri("https://other-service.example.com/endpoint")
        .retrieve()
        .bodyToMono(String.class)
```

Kotlin

```
this.rest.get()
        .uri("https://other-service.example.com/endpoint")
        .retrieve()
        .bodyToMono<String>()
```

Will invoke the `https://other-service.example.com/endpoint`, adding the bearer token `Authorization` header for you.

In places where you need to override this behavior, it's a simple matter of supplying the header yourself, like so:

Java

```
this.rest.get()
        .uri("https://other-service.example.com/endpoint")
        .headers(headers -> headers.setBearerAuth(overridingToken))
        .retrieve()
        .bodyToMono(String.class)
```

Kotlin

```
rest.get()
        .uri("https://other-service.example.com/endpoint")
        .headers { it.setBearerAuth(overridingToken) }
        .retrieve()
        .bodyToMono<String>()
```

In this case, the filter will fall back and simply forward the request onto the rest of the web filter chain.

Unlike the OAuth 2.0 Client filter function, this filter function makes no attempt to renew the token, should it be expired. To obtain this level of support, please use the OAuth 2.0 Client filter.

Chapter 25. @RegisteredOAuth2AuthorizedClient

Spring Security allows resolving an access token using `@RegisteredOAuth2AuthorizedClient`.

A working example can be found in **OAuth 2.0 WebClient WebFlux sample**.

After configuring Spring Security for OAuth2 Login or as an OAuth2 Client, an `OAuth2AuthorizedClient` can be resolved using the following:

Java

```
@GetMapping("/explicit")
Mono<String> explicit(@RegisteredOAuth2AuthorizedClient("client-id")
OAuth2AuthorizedClient authorizedClient) {
    // ...
}
```

Kotlin

```
@GetMapping("/explicit")
fun explicit(@RegisteredOAuth2AuthorizedClient("client-id") authorizedClient:
OAuth2AuthorizedClient?): Mono<String> {
    // ...
}
```

This integrates into Spring Security to provide the following features:

- Spring Security will automatically refresh expired tokens (if a refresh token is present)
- If an access token is requested and not present, Spring Security will automatically request the access token.
    - For `authorization_code` this involves performing the redirect and then replaying the original request
    - For `client_credentials` the token is simply requested and saved

If the user authenticated using `oauth2Login()`, then the `client-id` is optional. For example, the following would work:

Java

```
@GetMapping("/implicit")
Mono<String> implicit(@RegisteredOAuth2AuthorizedClient OAuth2AuthorizedClient
authorizedClient) {
    // ...
}
```

Kotlin

```
@GetMapping("/implicit")
fun implicit(@RegisteredOAuth2AuthorizedClient authorizedClient:
OAuth2AuthorizedClient?): Mono<String> {
    // ...
}
```

This is convenient if the user always authenticates with OAuth2 Login and an access token from the same authorization server is needed.

Chapter 26. Reactive X.509 Authentication

Similar to Servlet X.509 authentication, reactive x509 authentication filter allows extracting an authentication token from a certificate provided by a client.

Below is an example of a reactive x509 security configuration:

Java

```
@Bean
public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {
    http
        .x509(withDefaults())
        .authorizeExchange(exchanges -> exchanges
            .anyExchange().permitAll()
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun securityWebFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        x509 { }
        authorizeExchange {
            authorize(anyExchange, authenticated)
        }
    }
}
```

In the configuration above, when neither `principalExtractor` nor `authenticationManager` is provided defaults will be used. The default principal extractor is `SubjectDnX509PrincipalExtractor` which extracts the CN (common name) field from a certificate provided by a client. The default authentication manager is `ReactivePreAuthenticatedAuthenticationManager` which performs user account validation, checking that user account with a name extracted by `principalExtractor` exists and it is not locked, disabled, or expired.

The next example demonstrates how these defaults can be overridden.

Java

```
@Bean
public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {
    SubjectDnX509PrincipalExtractor principalExtractor =
            new SubjectDnX509PrincipalExtractor();

    principalExtractor.setSubjectDnRegex("OU=(.*?)(?:,|$)");

    ReactiveAuthenticationManager authenticationManager = authentication -> {
        authentication.setAuthenticated("Trusted Org
Unit".equals(authentication.getName()));
        return Mono.just(authentication);
    };

    http
        .x509(x509 -> x509
            .principalExtractor(principalExtractor)
            .authenticationManager(authenticationManager)
        )
        .authorizeExchange(exchanges -> exchanges
            .anyExchange().authenticated()
        );
    return http.build();
}
```

Kotlin

```
@Bean
fun securityWebFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain? {
    val customPrincipalExtractor = SubjectDnX509PrincipalExtractor()
    customPrincipalExtractor.setSubjectDnRegex("OU=(.*?)(?:,|$)")
    val customAuthenticationManager = ReactiveAuthenticationManager {
authentication: Authentication ->
        authentication.isAuthenticated = "Trusted Org Unit" == authentication.name
        Mono.just(authentication)
    }
    return http {
        x509 {
            principalExtractor = customPrincipalExtractor
            authenticationManager = customAuthenticationManager
        }
        authorizeExchange {
            authorize(anyExchange, authenticated)
        }
    }
}
```

In this example, a username is extracted from the OU field of a client certificate instead of CN, and account lookup using `ReactiveUserDetailsService` is not performed at all. Instead, if the provided certificate issued to an OU named "Trusted Org Unit", a request will be authenticated.

For an example of configuring Netty and `WebClient` or `curl` command-line tool to use mutual TLS and enable X.509 authentication, please refer to https://github.com/spring-projects/spring-security-samples/tree/main/servlet/java-configuration/authentication/x509.

Chapter 27. WebClient

The following documentation is for use within Reactive environments. For Servlet environments, refer to WebClient for Servlet environments.

Spring Framework has built in support for setting a Bearer token.

Java

```
webClient.get()
    .headers(h -> h.setBearerAuth(token))
    ...
```

Kotlin

```
webClient.get()
    .headers { it.setBearerAuth(token) }
    ...
```

Spring Security builds on this support to provide additional benefits:

- Spring Security will automatically refresh expired tokens (if a refresh token is present)
- If an access token is requested and not present, Spring Security will automatically request the access token.
    - For authorization_code this involves performing the redirect and then replaying the original request
    - For client_credentials the token is simply requested and saved
- Support for the ability to transparently include the current OAuth token or explicitly select which token should be used.

27.1. WebClient OAuth2 Setup

The first step is ensuring to setup the `WebClient` correctly. An example of setting up `WebClient` in a fully reactive environment can be found below:

Java

```
@Bean
WebClient webClient(ReactiveClientRegistrationRepository clientRegistrations,
        ServerOAuth2AuthorizedClientRepository authorizedClients) {
    ServerOAuth2AuthorizedClientExchangeFilterFunction oauth =
            new
ServerOAuth2AuthorizedClientExchangeFilterFunction(clientRegistrations,
authorizedClients);
    // (optional) explicitly opt into using the oauth2Login to provide an access
token implicitly
    // oauth.setDefaultOAuth2AuthorizedClient(true);
    // (optional) set a default ClientRegistration.registrationId
    // oauth.setDefaultClientRegistrationId("client-registration-id");
    return WebClient.builder()
            .filter(oauth)
            .build();
}
```

Kotlin

```
@Bean
fun webClient(clientRegistrations: ReactiveClientRegistrationRepository,
              authorizedClients: ServerOAuth2AuthorizedClientRepository):
WebClient {
    val oauth =
ServerOAuth2AuthorizedClientExchangeFilterFunction(clientRegistrations,
authorizedClients)
    // (optional) explicitly opt into using the oauth2Login to provide an access
token implicitly
    // oauth.setDefaultOAuth2AuthorizedClient(true)
    // (optional) set a default ClientRegistration.registrationId
    // oauth.setDefaultClientRegistrationId("client-registration-id")
    return WebClient.builder()
            .filter(oauth)
            .build()
}
```

27.2. Implicit OAuth2AuthorizedClient

If we set `defaultOAuth2AuthorizedClient` to `true` in our setup and the user authenticated with oauth2Login (i.e. OIDC), then the current authentication is used to automatically provide the access token. Alternatively, if we set `defaultClientRegistrationId` to a valid `ClientRegistration` id, that registration is used to provide the access token. This is convenient, but in environments where not all endpoints should get the access token, it is dangerous (you might provide the wrong access token to an endpoint).

Java

```
Mono<String> body = this.webClient
        .get()
        .uri(this.uri)
        .retrieve()
        .bodyToMono(String.class);
```

Kotlin

```
val body: Mono<String> = webClient
        .get()
        .uri(this.uri)
        .retrieve()
        .bodyToMono()
```

27.3. Explicit OAuth2AuthorizedClient

The `OAuth2AuthorizedClient` can be explicitly provided by setting it on the requests attributes. In the example below we resolve the `OAuth2AuthorizedClient` using Spring WebFlux or Spring MVC argument resolver support. However, it does not matter how the `OAuth2AuthorizedClient` is resolved.

Java

```
@GetMapping("/explicit")
Mono<String> explicit(@RegisteredOAuth2AuthorizedClient("client-id")
OAuth2AuthorizedClient authorizedClient) {
    return this.webClient
            .get()
            .uri(this.uri)
            .attributes(oauth2AuthorizedClient(authorizedClient))
            .retrieve()
            .bodyToMono(String.class);
}
```

Kotlin

```
@GetMapping("/explicit")
fun explicit(@RegisteredOAuth2AuthorizedClient("client-id") authorizedClient:
OAuth2AuthorizedClient?): Mono<String> {
    return this.webClient
            .get()
            .uri(uri)
            .attributes(oauth2AuthorizedClient(authorizedClient))
            .retrieve()
            .bodyToMono()
}
```

27.4. clientRegistrationId

Alternatively, it is possible to specify the `clientRegistrationId` on the request attributes and the `WebClient` will attempt to lookup the `OAuth2AuthorizedClient`. If it is not found, one will automatically be acquired.

Java

```
Mono<String> body = this.webClient
        .get()
        .uri(this.uri)
        .attributes(clientRegistrationId("client-id"))
        .retrieve()
        .bodyToMono(String.class);
```

Kotlin

```
val body: Mono<String> = this.webClient
        .get()
        .uri(uri)
        .attributes(clientRegistrationId("client-id"))
        .retrieve()
        .bodyToMono()
```

Chapter 28. EnableReactiveMethodSecurity

Spring Security supports method security using Reactor's Context which is setup using `ReactiveSecurityContextHolder`. For example, this demonstrates how to retrieve the currently logged in user's message.

For this to work the return type of the method must be a `org.reactivestreams.Publisher` (i.e. `Mono`/`Flux`) or the function must be a Kotlin coroutine function. This is necessary to integrate with Reactor's `Context`.

Java

```
Authentication authentication = new TestingAuthenticationToken("user", "password",
"ROLE_USER");

Mono<String> messageByUsername = ReactiveSecurityContextHolder.getContext()
    .map(SecurityContext::getAuthentication)
    .map(Authentication::getName)
    .flatMap(this::findMessageByUsername)
    // In a WebFlux application the `subscriberContext` is automatically setup
using `ReactorContextWebFilter`

.subscriberContext(ReactiveSecurityContextHolder.withAuthentication(authentication
));

StepVerifier.create(messageByUsername)
    .expectNext("Hi user")
    .verifyComplete();
```

Kotlin

```
val authentication: Authentication = TestingAuthenticationToken("user",
"password", "ROLE_USER")

val messageByUsername: Mono<String> = ReactiveSecurityContextHolder.getContext()
    .map(SecurityContext::getAuthentication)
    .map(Authentication::getName)
    .flatMap(this::findMessageByUsername) // In a WebFlux application the
`subscriberContext` is automatically setup using `ReactorContextWebFilter`

.subscriberContext(ReactiveSecurityContextHolder.withAuthentication(authentication
))

StepVerifier.create(messageByUsername)
    .expectNext("Hi user")
    .verifyComplete()
```

with `this::findMessageByUsername` defined as:

Java

```
Mono<String> findMessageByUsername(String username) {
    return Mono.just("Hi " + username);
}
```

Kotlin

```
fun findMessageByUsername(username: String): Mono<String> {
    return Mono.just("Hi $username")
}
```

Below is a minimal method security configuration when using method security in reactive applications.

Java

```
@EnableReactiveMethodSecurity
public class SecurityConfig {
    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        User.UserBuilder userBuilder = User.withDefaultPasswordEncoder();
        UserDetails rob = userBuilder.username("rob")
            .password("rob")
            .roles("USER")
            .build();
        UserDetails admin = userBuilder.username("admin")
            .password("admin")
            .roles("USER","ADMIN")
            .build();
        return new MapReactiveUserDetailsService(rob, admin);
    }
}
```

Kotlin

```
@EnableReactiveMethodSecurity
class SecurityConfig {
    @Bean
    fun userDetailsService(): MapReactiveUserDetailsService {
        val userBuilder: User.UserBuilder = User.withDefaultPasswordEncoder()
        val rob = userBuilder.username("rob")
            .password("rob")
            .roles("USER")
            .build()
        val admin = userBuilder.username("admin")
            .password("admin")
            .roles("USER", "ADMIN")
            .build()
        return MapReactiveUserDetailsService(rob, admin)
    }
}
```

Consider the following class:

Java

```
@Component
public class HelloWorldMessageService {
    @PreAuthorize("hasRole('ADMIN')")
    public Mono<String> findMessage() {
        return Mono.just("Hello World!");
    }
}
```

Kotlin

```
@Component
class HelloWorldMessageService {
    @PreAuthorize("hasRole('ADMIN')")
    fun findMessage(): Mono<String> {
        return Mono.just("Hello World!")
    }
}
```

Or, the following class using Kotlin coroutines:

Kotlin

```
@Component
class HelloWorldMessageService {
    @PreAuthorize("hasRole('ADMIN')")
    suspend fun findMessage(): String {
        delay(10)
        return "Hello World!"
    }
}
```

Combined with our configuration above, `@PreAuthorize("hasRole('ADMIN')")` will ensure that `findByMessage` is only invoked by a user with the role `ADMIN`. It is important to note that any of the expressions in standard method security work for `@EnableReactiveMethodSecurity`. However, at this time we only support return type of `Boolean` or `boolean` of the expression. This means that the expression must not block.

When integrating with WebFlux Security, the Reactor Context is automatically established by Spring Security according to the authenticated user.

Java

```
@EnableWebFluxSecurity
@EnableReactiveMethodSecurity
public class SecurityConfig {

    @Bean
    SecurityWebFilterChain springWebFilterChain(ServerHttpSecurity http) throws
Exception {
        return http
            // Demonstrate that method security works
            // Best practice to use both for defense in depth
            .authorizeExchange(exchanges -> exchanges
                .anyExchange().permitAll()
            )
            .httpBasic(withDefaults())
            .build();
    }

    @Bean
    MapReactiveUserDetailsService userDetailsService() {
        User.UserBuilder userBuilder = User.withDefaultPasswordEncoder();
        UserDetails rob = userBuilder.username("rob")
            .password("rob")
            .roles("USER")
            .build();
        UserDetails admin = userBuilder.username("admin")
            .password("admin")
            .roles("USER","ADMIN")
            .build();
        return new MapReactiveUserDetailsService(rob, admin);
    }
}
```

Kotlin

```
@EnableWebFluxSecurity
@EnableReactiveMethodSecurity
class SecurityConfig {
    @Bean
    open fun springWebFilterChain(http: ServerHttpSecurity):
SecurityWebFilterChain {
        return http {
            authorizeExchange {
                authorize(anyExchange, permitAll)
            }
            httpBasic { }
        }
    }

    @Bean
    fun userDetailsService(): MapReactiveUserDetailsService {
        val userBuilder: User.UserBuilder = User.withDefaultPasswordEncoder()
        val rob = userBuilder.username("rob")
            .password("rob")
            .roles("USER")
            .build()
        val admin = userBuilder.username("admin")
            .password("admin")
            .roles("USER", "ADMIN")
            .build()
        return MapReactiveUserDetailsService(rob, admin)
    }
}
```

You can find a complete sample in hellowebflux-method

Chapter 29. CORS

Spring Framework provides first class support for CORS. CORS must be processed before Spring Security because the pre-flight request will not contain any cookies (i.e. the `JSESSIONID`). If the request does not contain any cookies and Spring Security is first, the request will determine the user is not authenticated (since there are no cookies in the request) and reject it.

The easiest way to ensure that CORS is handled first is to use the `CorsWebFilter`. Users can integrate the `CorsWebFilter` with Spring Security by providing a `CorsConfigurationSource`. For example, the following will integrate CORS support within Spring Security:

Java

```
@Bean
CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.setAllowedOrigins(Arrays.asList("https://example.com"));
    configuration.setAllowedMethods(Arrays.asList("GET","POST"));
    UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", configuration);
    return source;
}
```

Kotlin

```
@Bean
fun corsConfigurationSource(): CorsConfigurationSource {
    val configuration = CorsConfiguration()
    configuration.allowedOrigins = listOf("https://example.com")
    configuration.allowedMethods = listOf("GET", "POST")
    val source = UrlBasedCorsConfigurationSource()
    source.registerCorsConfiguration("/**", configuration)
    return source
}
```

The following will disable the CORS integration within Spring Security:

Java

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    http
        // ...
        .cors(cors -> cors.disable());
    return http.build();
}
```

Kotlin

```
@Bean
fun springSecurityFilterChain(http: ServerHttpSecurity): SecurityWebFilterChain {
    return http {
        // ...
        cors {
            disable()
        }
    }
}
```

Chapter 30. Reactive Test Support

30.1. Testing Reactive Method Security

For example, we can test our example from EnableReactiveMethodSecurity using the same setup and annotations we did in Testing Method Security. Here is a minimal sample of what we can do:

Java

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = HelloWebfluxMethodApplication.class)
public class HelloWorldMessageServiceTests {
    @Autowired
    HelloWorldMessageService messages;

    @Test
    public void messagesWhenNotAuthenticatedThenDenied() {
        StepVerifier.create(this.messages.findMessage())
            .expectError(AccessDeniedException.class)
            .verify();
    }

    @Test
    @WithMockUser
    public void messagesWhenUserThenDenied() {
        StepVerifier.create(this.messages.findMessage())
            .expectError(AccessDeniedException.class)
            .verify();
    }

    @Test
    @WithMockUser(roles = "ADMIN")
    public void messagesWhenAdminThenOk() {
        StepVerifier.create(this.messages.findMessage())
            .expectNext("Hello World!")
            .verifyComplete();
    }
}
```

Kotlin

```
@RunWith(SpringRunner::class)
@ContextConfiguration(classes = [HelloWebfluxMethodApplication::class])
class HelloWorldMessageServiceTests {
    @Autowired
    lateinit var messages: HelloWorldMessageService

    @Test
    fun messagesWhenNotAuthenticatedThenDenied() {
        StepVerifier.create(messages.findMessage())
            .expectError(AccessDeniedException::class.java)
            .verify()
    }

    @Test
    @WithMockUser
    fun messagesWhenUserThenDenied() {
        StepVerifier.create(messages.findMessage())
            .expectError(AccessDeniedException::class.java)
            .verify()
    }

    @Test
    @WithMockUser(roles = ["ADMIN"])
    fun messagesWhenAdminThenOk() {
        StepVerifier.create(messages.findMessage())
            .expectNext("Hello World!")
            .verifyComplete()
    }
}
```

30.2. WebTestClientSupport

Spring Security provides integration with `WebTestClient`. The basic setup looks like this:

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = HelloWebfluxMethodApplication.class)
public class HelloWebfluxMethodApplicationTests {
    @Autowired
    ApplicationContext context;

    WebTestClient rest;

    @Before
    public void setup() {
        this.rest = WebTestClient
            .bindToApplicationContext(this.context)
            // add Spring Security test Support
            .apply(springSecurity())
            .configureClient()
            .filter(basicAuthentication())
            .build();
    }
    // ...
}
```

30.2.1. Authentication

After applying the Spring Security support to `WebTestClient` we can use either annotations or `mutateWith` support. For example:

Java

```
@Test
public void messageWhenNotAuthenticated() throws Exception {
    this.rest
        .get()
        .uri("/message")
        .exchange()
        .expectStatus().isUnauthorized();
}

// --- WithMockUser ---

@Test
@WithMockUser
public void messageWhenWithMockUserThenForbidden() throws Exception {
    this.rest
        .get()
        .uri("/message")
        .exchange()
        .expectStatus().isEqualTo(HttpStatus.FORBIDDEN);
}

@Test
@WithMockUser(roles = "ADMIN")
public void messageWhenWithMockAdminThenOk() throws Exception {
    this.rest
        .get()
        .uri("/message")
        .exchange()
        .expectStatus().isOk()
        .expectBody(String.class).isEqualTo("Hello World!");
}

// --- mutateWith mockUser ---

@Test
public void messageWhenMutateWithMockUserThenForbidden() throws Exception {
    this.rest
        .mutateWith(mockUser())
        .get()
        .uri("/message")
        .exchange()
        .expectStatus().isEqualTo(HttpStatus.FORBIDDEN);
}

@Test
public void messageWhenMutateWithMockAdminThenOk() throws Exception {
    this.rest
        .mutateWith(mockUser().roles("ADMIN"))
```

```
        .get()
        .uri("/message")
        .exchange()
        .expectStatus().isOk()
        .expectBody(String.class).isEqualTo("Hello World!");
}
```

Kotlin

```
import org.springframework.test.web.reactive.server.expectBody

//...

@Test
@WithMockUser
fun messageWhenWithMockUserThenForbidden() {
    this.rest.get().uri("/message")
        .exchange()
        .expectStatus().isEqualTo(HttpStatus.FORBIDDEN)
}

@Test
@WithMockUser(roles = ["ADMIN"])
fun messageWhenWithMockAdminThenOk() {
    this.rest.get().uri("/message")
        .exchange()
        .expectStatus().isOk
        .expectBody<String>().isEqualTo("Hello World!")

}

// --- mutateWith mockUser ---

@Test
fun messageWhenMutateWithMockUserThenForbidden() {
    this.rest
        .mutateWith(mockUser())
        .get().uri("/message")
        .exchange()
        .expectStatus().isEqualTo(HttpStatus.FORBIDDEN)
}

@Test
fun messageWhenMutateWithMockAdminThenOk() {
    this.rest
        .mutateWith(mockUser().roles("ADMIN"))
        .get().uri("/message")
        .exchange()
        .expectStatus().isOk
        .expectBody<String>().isEqualTo("Hello World!")
}
```

30.2.2. CSRF Support

Spring Security also provides support for CSRF testing with `WebTestClient`. For example:

Java

```
this.rest
    // provide a valid CSRF token
    .mutateWith(csrf())
    .post()
    .uri("/login")
    ...
```

Kotlin

```
this.rest
    // provide a valid CSRF token
    .mutateWith(csrf())
    .post()
    .uri("/login")
    ...
```

30.2.3. Testing OAuth 2.0

When it comes to OAuth 2.0, the same principles covered earlier still apply: Ultimately, it depends on what your method under test is expecting to be in the `SecurityContextHolder`.

For example, for a controller that looks like this:

Java

```
@GetMapping("/endpoint")
public Mono<String> foo(Principal user) {
    return Mono.just(user.getName());
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(user: Principal): Mono<String> {
    return Mono.just(user.name)
}
```

There's nothing OAuth2-specific about it, so you will likely be able to simply use `@WithMockUser` and be fine.

But, in cases where your controllers are bound to some aspect of Spring Security's OAuth 2.0 support, like the following:

Java

```
@GetMapping("/endpoint")
public Mono<String> foo(@AuthenticationPrincipal OidcUser user) {
    return Mono.just(user.getIdToken().getSubject());
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(@AuthenticationPrincipal user: OidcUser): Mono<String> {
    return Mono.just(user.idToken.subject)
}
```

then Spring Security's test support can come in handy.

30.2.4. Testing OIDC Login

Testing the method above with `WebTestClient` would require simulating some kind of grant flow with an authorization server. Certainly this would be a daunting task, which is why Spring Security ships with support for removing this boilerplate.

For example, we can tell Spring Security to include a default `OidcUser` using the `SecurityMockServerConfigurers#oidcLogin` method, like so:

Java

```
client
    .mutateWith(mockOidcLogin()).get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockOidcLogin())
    .get().uri("/endpoint")
    .exchange()
```

What this will do is configure the associated `MockServerRequest` with an `OidcUser` that includes a simple `OidcIdToken`, `OidcUserInfo`, and `Collection` of granted authorities.

Specifically, it will include an `OidcIdToken` with a `sub` claim set to `user`:

Java

```
assertThat(user.getIdToken().getClaim("sub")).isEqualTo("user");
```

Kotlin

```
assertThat(user.idToken.getClaim<String>("sub")).isEqualTo("user")
```

an `OidcUserInfo` with no claims set:

Java

```
assertThat(user.getUserInfo().getClaims()).isEmpty();
```

Kotlin

```
assertThat(user.userInfo.claims).isEmpty()
```

and a `Collection` of authorities with just one authority, `SCOPE_read`:

Java

```
assertThat(user.getAuthorities()).hasSize(1);
assertThat(user.getAuthorities()).containsExactly(new
SimpleGrantedAuthority("SCOPE_read"));
```

Kotlin

```
assertThat(user.authorities).hasSize(1)
assertThat(user.authorities).containsExactly(SimpleGrantedAuthority("SCOPE_read"))
```

Spring Security does the necessary work to make sure that the `OidcUser` instance is available for the `@AuthenticationPrincipal` annotation.

Further, it also links that `OidcUser` to a simple instance of `OAuth2AuthorizedClient` that it deposits into a mock `ServerOAuth2AuthorizedClientRepository`. This can be handy if your tests use the `@RegisteredOAuth2AuthorizedClient` annotation..

Configuring Authorities

In many circumstances, your method is protected by filter or method security and needs your `Authentication` to have certain granted authorities to allow the request.

In this case, you can supply what granted authorities you need using the `authorities()` method:

Java

```
client
    .mutateWith(mockOidcLogin()
        .authorities(new SimpleGrantedAuthority("SCOPE_message:read"))
    )
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockOidcLogin()
        .authorities(SimpleGrantedAuthority("SCOPE_message:read"))
    )
    .get().uri("/endpoint").exchange()
```

Configuring Claims

And while granted authorities are quite common across all of Spring Security, we also have claims in the case of OAuth 2.0.

Let's say, for example, that you've got a `user_id` claim that indicates the user's id in your system. You might access it like so in a controller:

Java

```
@GetMapping("/endpoint")
public Mono<String> foo(@AuthenticationPrincipal OidcUser oidcUser) {
    String userId = oidcUser.getIdToken().getClaim("user_id");
    // ...
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(@AuthenticationPrincipal oidcUser: OidcUser): Mono<String> {
    val userId = oidcUser.idToken.getClaim<String>("user_id")
    // ...
}
```

In that case, you'd want to specify that claim with the `idToken()` method:

Java

```
client
    .mutateWith(mockOidcLogin()
        .idToken(token -> token.claim("user_id", "1234"))
    )
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockOidcLogin()
        .idToken { token -> token.claim("user_id", "1234") }
    )
    .get().uri("/endpoint").exchange()
```

since `OidcUser` collects its claims from `OidcIdToken`.

Additional Configurations

There are additional methods, too, for further configuring the authentication; it simply depends on what data your controller expects:

- `userInfo(OidcUserInfo.Builder)` - For configuring the `OidcUserInfo` instance
- `clientRegistration(ClientRegistration)` - For configuring the associated `OAuth2AuthorizedClient` with a given `ClientRegistration`
- `oidcUser(OidcUser)` - For configuring the complete `OidcUser` instance

That last one is handy if you: 1. Have your own implementation of `OidcUser`, or 2. Need to change the name attribute

For example, let's say that your authorization server sends the principal name in the `user_name` claim instead of the `sub` claim. In that case, you can configure an `OidcUser` by hand:

Java

```
OidcUser oidcUser = new DefaultOidcUser(
        AuthorityUtils.createAuthorityList("SCOPE_message:read"),
        OidcIdToken.withTokenValue("id-token").claim("user_name",
"foo_user").build(),
        "user_name");

client
    .mutateWith(mockOidcLogin().oidcUser(oidcUser))
    .get().uri("/endpoint").exchange();
```

Kotlin

```
val oidcUser: OidcUser = DefaultOidcUser(
    AuthorityUtils.createAuthorityList("SCOPE_message:read"),
    OidcIdToken.withTokenValue("id-token").claim("user_name", "foo_user").build(),
    "user_name"
)

client
    .mutateWith(mockOidcLogin().oidcUser(oidcUser))
    .get().uri("/endpoint").exchange()
```

30.2.5. Testing OAuth 2.0 Login

As with testing OIDC login, testing OAuth 2.0 Login presents a similar challenge of mocking a grant flow. And because of that, Spring Security also has test support for non-OIDC use cases.

Let's say that we've got a controller that gets the logged-in user as an `OAuth2User`:

Java

```
@GetMapping("/endpoint")
public Mono<String> foo(@AuthenticationPrincipal OAuth2User oauth2User) {
    return Mono.just(oauth2User.getAttribute("sub"));
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(@AuthenticationPrincipal oauth2User: OAuth2User): Mono<String> {
    return Mono.just(oauth2User.getAttribute("sub"))
}
```

In that case, we can tell Spring Security to include a default `OAuth2User` using the `SecurityMockServerConfigurers#oauth2User` method, like so:

Java

```
client
    .mutateWith(mockOAuth2Login())
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockOAuth2Login())
    .get().uri("/endpoint").exchange()
```

What this will do is configure the associated `MockServerRequest` with an `OAuth2User` that includes a simple `Map` of attributes and `Collection` of granted authorities.

Specifically, it will include a `Map` with a key/value pair of `sub`/`user`:

Java

```
assertThat((String) user.getAttribute("sub")).isEqualTo("user");
```

Kotlin

```
assertThat(user.getAttribute<String>("sub")).isEqualTo("user")
```

and a `Collection` of authorities with just one authority, `SCOPE_read`:

Java

```
assertThat(user.getAuthorities()).hasSize(1);
assertThat(user.getAuthorities()).containsExactly(new
SimpleGrantedAuthority("SCOPE_read"));
```

Kotlin

```
assertThat(user.authorities).hasSize(1)
assertThat(user.authorities).containsExactly(SimpleGrantedAuthority("SCOPE_read"))
```

Spring Security does the necessary work to make sure that the `OAuth2User` instance is available for the `@AuthenticationPrincipal` annotation.

Further, it also links that `OAuth2User` to a simple instance of `OAuth2AuthorizedClient` that it deposits in a mock `ServerOAuth2AuthorizedClientRepository`. This can be handy if your tests use the `@RegisteredOAuth2AuthorizedClient` annotation.

Configuring Authorities

In many circumstances, your method is protected by filter or method security and needs your `Authentication` to have certain granted authorities to allow the request.

In this case, you can supply what granted authorities you need using the `authorities()` method:

Java

```
client
    .mutateWith(mockOAuth2Login()
        .authorities(new SimpleGrantedAuthority("SCOPE_message:read"))
    )
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockOAuth2Login()
        .authorities(SimpleGrantedAuthority("SCOPE_message:read"))
    )
    .get().uri("/endpoint").exchange()
```

Configuring Claims

And while granted authorities are quite common across all of Spring Security, we also have claims in the case of OAuth 2.0.

Let's say, for example, that you've got a `user_id` attribute that indicates the user's id in your system. You might access it like so in a controller:

Java

```
@GetMapping("/endpoint")
public Mono<String> foo(@AuthenticationPrincipal OAuth2User oauth2User) {
    String userId = oauth2User.getAttribute("user_id");
    // ...
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(@AuthenticationPrincipal oauth2User: OAuth2User): Mono<String> {
    val userId = oauth2User.getAttribute<String>("user_id")
    // ...
}
```

In that case, you'd want to specify that attribute with the `attributes()` method:

Java

```
client
    .mutateWith(mockOAuth2Login()
        .attributes(attrs -> attrs.put("user_id", "1234"))
    )
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockOAuth2Login()
        .attributes { attrs -> attrs["user_id"] = "1234" }
    )
    .get().uri("/endpoint").exchange()
```

Additional Configurations

There are additional methods, too, for further configuring the authentication; it simply depends on what data your controller expects:

- `clientRegistration(ClientRegistration)` - For configuring the associated `OAuth2AuthorizedClient` with a given `ClientRegistration`
- `oauth2User(OAuth2User)` - For configuring the complete `OAuth2User` instance

That last one is handy if you: 1. Have your own implementation of `OAuth2User`, or 2. Need to change the name attribute

For example, let's say that your authorization server sends the principal name in the `user_name` claim instead of the `sub` claim. In that case, you can configure an `OAuth2User` by hand:

Java

```
OAuth2User oauth2User = new DefaultOAuth2User(
        AuthorityUtils.createAuthorityList("SCOPE_message:read"),
        Collections.singletonMap("user_name", "foo_user"),
        "user_name");

client
    .mutateWith(mockOAuth2Login().oauth2User(oauth2User))
    .get().uri("/endpoint").exchange();
```

Kotlin

```
val oauth2User: OAuth2User = DefaultOAuth2User(
    AuthorityUtils.createAuthorityList("SCOPE_message:read"),
    mapOf(Pair("user_name", "foo_user")),
    "user_name"
)

client
    .mutateWith(mockOAuth2Login().oauth2User(oauth2User))
    .get().uri("/endpoint").exchange()
```

30.2.6. Testing OAuth 2.0 Clients

Independent of how your user authenticates, you may have other tokens and client registrations that are in play for the request you are testing. For example, your controller may be relying on the client credentials grant to get a token that isn't associated with the user at all:

Java

```
@GetMapping("/endpoint")
public Mono<String> foo(@RegisteredOAuth2AuthorizedClient("my-app")
OAuth2AuthorizedClient authorizedClient) {
    return this.webClient.get()
        .attributes(oauth2AuthorizedClient(authorizedClient))
        .retrieve()
        .bodyToMono(String.class);
}
```

Kotlin

```
import org.springframework.web.reactive.function.client.bodyToMono

// ...

@GetMapping("/endpoint")
fun foo(@RegisteredOAuth2AuthorizedClient("my-app") authorizedClient:
OAuth2AuthorizedClient?): Mono<String> {
    return this.webClient.get()
        .attributes(oauth2AuthorizedClient(authorizedClient))
        .retrieve()
        .bodyToMono()
}
```

Simulating this handshake with the authorization server could be cumbersome. Instead, you can use `SecurityMockServerConfigurers#oauth2Client` to add a `OAuth2AuthorizedClient` into a mock `ServerOAuth2AuthorizedClientRepository`:

Java

```
client
    .mutateWith(mockOAuth2Client("my-app"))
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockOAuth2Client("my-app"))
    .get().uri("/endpoint").exchange()
```

What this will do is create an `OAuth2AuthorizedClient` that has a simple `ClientRegistration`, `OAuth2AccessToken`, and resource owner name.

Specifically, it will include a `ClientRegistration` with a client id of "test-client" and client secret of "test-secret":

Java

```
assertThat(authorizedClient.getClientRegistration().getClientId()).isEqualTo("test
-client");
assertThat(authorizedClient.getClientRegistration().getClientSecret()).isEqualTo("
test-secret");
```

Kotlin

```
assertThat(authorizedClient.clientRegistration.clientId).isEqualTo("test-client")
assertThat(authorizedClient.clientRegistration.clientSecret).isEqualTo("test-
secret")
```

a resource owner name of "user":

Java

```
assertThat(authorizedClient.getPrincipalName()).isEqualTo("user");
```

Kotlin

```
assertThat(authorizedClient.principalName).isEqualTo("user")
```

and an `OAuth2AccessToken` with just one scope, `read`:

Java

```
assertThat(authorizedClient.getAccessToken().getScopes()).hasSize(1);
assertThat(authorizedClient.getAccessToken().getScopes()).containsExactly("read");
```

Kotlin

```
assertThat(authorizedClient.accessToken.scopes).hasSize(1)
assertThat(authorizedClient.accessToken.scopes).containsExactly("read")
```

The client can then be retrieved as normal using `@RegisteredOAuth2AuthorizedClient` in a controller method.

Configuring Scopes

In many circumstances, the OAuth 2.0 access token comes with a set of scopes. If your controller inspects these, say like so:

Java

```
@GetMapping("/endpoint")
public Mono<String> foo(@RegisteredOAuth2AuthorizedClient("my-app")
OAuth2AuthorizedClient authorizedClient) {
    Set<String> scopes = authorizedClient.getAccessToken().getScopes();
    if (scopes.contains("message:read")) {
        return this.webClient.get()
            .attributes(oauth2AuthorizedClient(authorizedClient))
            .retrieve()
            .bodyToMono(String.class);
    }
    // ...
}
```

Kotlin

```
import org.springframework.web.reactive.function.client.bodyToMono

// ...

@GetMapping("/endpoint")
fun foo(@RegisteredOAuth2AuthorizedClient("my-app") authorizedClient:
OAuth2AuthorizedClient): Mono<String> {
    val scopes = authorizedClient.accessToken.scopes
    if (scopes.contains("message:read")) {
        return webClient.get()
            .attributes(oauth2AuthorizedClient(authorizedClient))
            .retrieve()
            .bodyToMono()
    }
    // ...
}
```

then you can configure the scope using the `accessToken()` method:

Java

```
client
    .mutateWith(mockOAuth2Client("my-app")
        .accessToken(new OAuth2AccessToken(BEARER, "token", null, null,
Collections.singleton("message:read")))
    )
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockOAuth2Client("my-app")
        .accessToken(OAuth2AccessToken(BEARER, "token", null, null,
setOf("message:read")))
)
.get().uri("/endpoint").exchange()
```

Additional Configurations

There are additional methods, too, for further configuring the authentication; it simply depends on what data your controller expects:

- `principalName(String)` - For configuring the resource owner name
- `clientRegistration(Consumer<ClientRegistration.Builder>)` - For configuring the associated `ClientRegistration`
- `clientRegistration(ClientRegistration)` - For configuring the complete `ClientRegistration`

That last one is handy if you want to use a real `ClientRegistration`

For example, let's say that you are wanting to use one of your app's `ClientRegistration` definitions, as specified in your `application.yml`.

In that case, your test can autowire the `ReactiveClientRegistrationRepository` and look up the one your test needs:

Java

```
@Autowired
ReactiveClientRegistrationRepository clientRegistrationRepository;

// ...

client
    .mutateWith(mockOAuth2Client()

.clientRegistration(this.clientRegistrationRepository.findByRegistrationId("facebo
ok").block())
    )
    .get().uri("/exchange").exchange();
```

Kotlin

```
@Autowired
lateinit var clientRegistrationRepository: ReactiveClientRegistrationRepository

// ...

client
    .mutateWith(mockOAuth2Client()

.clientRegistration(this.clientRegistrationRepository.findByRegistrationId("facebo
ok").block())
    )
    .get().uri("/exchange").exchange()
```

30.2.7. Testing JWT Authentication

In order to make an authorized request on a resource server, you need a bearer token. If your resource server is configured for JWTs, then this would mean that the bearer token needs to be signed and then encoded according to the JWT specification. All of this can be quite daunting, especially when this isn't the focus of your test.

Fortunately, there are a number of simple ways that you can overcome this difficulty and allow your tests to focus on authorization and not on representing bearer tokens. We'll look at two of them now:

`mockJwt()` `WebTestClientConfigurer`

The first way is via a `WebTestClientConfigurer`. The simplest of these would look something like this:

Java

```
client
    .mutateWith(mockJwt()).get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockJwt()).get().uri("/endpoint").exchange()
```

What this will do is create a mock `Jwt`, passing it correctly through any authentication APIs so that it's available for your authorization mechanisms to verify.

By default, the `JWT` that it creates has the following characteristics:

```
{
  "headers" : { "alg" : "none" },
  "claims" : {
    "sub" : "user",
    "scope" : "read"
  }
}
```

And the resulting `Jwt`, were it tested, would pass in the following way:

Java

```
assertThat(jwt.getTokenValue()).isEqualTo("token");
assertThat(jwt.getHeaders().get("alg")).isEqualTo("none");
assertThat(jwt.getSubject()).isEqualTo("sub");
```

Kotlin

```
assertThat(jwt.tokenValue).isEqualTo("token")
assertThat(jwt.headers["alg"]).isEqualTo("none")
assertThat(jwt.subject).isEqualTo("sub")
```

These values can, of course be configured.

Any headers or claims can be configured with their corresponding methods:

Java

```
client
    .mutateWith(mockJwt().jwt(jwt -> jwt.header("kid", "one")
        .claim("iss", "https://idp.example.org")))
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockJwt().jwt { jwt -> jwt.header("kid", "one")
        .claim("iss", "https://idp.example.org")
    })
    .get().uri("/endpoint").exchange()
```

Java

```
client
    .mutateWith(mockJwt().jwt(jwt -> jwt.claims(claims ->
claims.remove("scope"))))
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockJwt().jwt { jwt ->
        jwt.claims { claims -> claims.remove("scope") }
    })
    .get().uri("/endpoint").exchange()
```

The `scope` and `scp` claims are processed the same way here as they are in a normal bearer token request. However, this can be overridden simply by providing the list of `GrantedAuthority` instances that you need for your test:

Java

```
client
    .mutateWith(mockJwt().authorities(new
SimpleGrantedAuthority("SCOPE_messages")))
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockJwt().authorities(SimpleGrantedAuthority("SCOPE_messages")))
    .get().uri("/endpoint").exchange()
```

Or, if you have a custom `Jwt` to `Collection<GrantedAuthority>` converter, you can also use that to derive the authorities:

Java

```
client
    .mutateWith(mockJwt().authorities(new MyConverter()))
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockJwt().authorities(MyConverter()))
    .get().uri("/endpoint").exchange()
```

You can also specify a complete `Jwt`, for which `Jwt.Builder` comes quite handy:

Java

```
Jwt jwt = Jwt.withTokenValue("token")
    .header("alg", "none")
    .claim("sub", "user")
    .claim("scope", "read")
    .build();

client
    .mutateWith(mockJwt().jwt(jwt))
    .get().uri("/endpoint").exchange();
```

Kotlin

```
val jwt: Jwt = Jwt.withTokenValue("token")
    .header("alg", "none")
    .claim("sub", "user")
    .claim("scope", "read")
    .build()

client
    .mutateWith(mockJwt().jwt(jwt))
    .get().uri("/endpoint").exchange()
```

`authentication()` `WebTestClientConfigurer`

The second way is by using the `authentication()` `Mutator`. Essentially, you can instantiate your own `JwtAuthenticationToken` and provide it in your test, like so:

Java

```
Jwt jwt = Jwt.withTokenValue("token")
    .header("alg", "none")
    .claim("sub", "user")
    .build();
Collection<GrantedAuthority> authorities =
AuthorityUtils.createAuthorityList("SCOPE_read");
JwtAuthenticationToken token = new JwtAuthenticationToken(jwt, authorities);

client
    .mutateWith(mockAuthentication(token))
    .get().uri("/endpoint").exchange();
```

Kotlin

```
val jwt = Jwt.withTokenValue("token")
    .header("alg", "none")
    .claim("sub", "user")
    .build()
val authorities: Collection<GrantedAuthority> =
AuthorityUtils.createAuthorityList("SCOPE_read")
val token = JwtAuthenticationToken(jwt, authorities)

client
    .mutateWith(mockAuthentication<JwtMutator>(token))
    .get().uri("/endpoint").exchange()
```

Note that as an alternative to these, you can also mock the `ReactiveJwtDecoder` bean itself with a `@MockBean` annotation.

30.2.8. Testing Opaque Token Authentication

Similar to JWTs, opaque tokens require an authorization server in order to verify their validity, which can make testing more difficult. To help with that, Spring Security has test support for opaque tokens.

Let's say that we've got a controller that retrieves the authentication as a `BearerTokenAuthentication`:

Java

```
@GetMapping("/endpoint")
public Mono<String> foo(BearerTokenAuthentication authentication) {
    return Mono.just((String) authentication.getTokenAttributes().get("sub"));
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(authentication: BearerTokenAuthentication): Mono<String?> {
    return Mono.just(authentication.tokenAttributes["sub"] as String?)
}
```

In that case, we can tell Spring Security to include a default `BearerTokenAuthentication` using the `SecurityMockServerConfigurers#opaqueToken` method, like so:

Java

```
client
    .mutateWith(mockOpaqueToken())
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockOpaqueToken())
    .get().uri("/endpoint").exchange()
```

What this will do is configure the associated `MockHttpServletRequest` with a `BearerTokenAuthentication` that includes a simple `OAuth2AuthenticatedPrincipal`, `Map` of attributes, and `Collection` of granted authorities.

Specifically, it will include a `Map` with a key/value pair of `sub`/`user`:

Java

```
assertThat((String) token.getTokenAttributes().get("sub")).isEqualTo("user");
```

Kotlin

```
assertThat(token.tokenAttributes["sub"] as String?).isEqualTo("user")
```

and a `Collection` of authorities with just one authority, `SCOPE_read`:

Java

```
assertThat(token.getAuthorities()).hasSize(1);
assertThat(token.getAuthorities()).containsExactly(new
SimpleGrantedAuthority("SCOPE_read"));
```

Kotlin

```
assertThat(token.authorities).hasSize(1)
assertThat(token.authorities).containsExactly(SimpleGrantedAuthority("SCOPE_read")
)
```

Spring Security does the necessary work to make sure that the `BearerTokenAuthentication` instance is available for your controller methods.

Configuring Authorities

In many circumstances, your method is protected by filter or method security and needs your `Authentication` to have certain granted authorities to allow the request.

In this case, you can supply what granted authorities you need using the `authorities()` method:

Java

```
client
    .mutateWith(mockOpaqueToken()
        .authorities(new SimpleGrantedAuthority("SCOPE_message:read"))
    )
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockOpaqueToken()
        .authorities(SimpleGrantedAuthority("SCOPE_message:read"))
    )
    .get().uri("/endpoint").exchange()
```

Configuring Claims

And while granted authorities are quite common across all of Spring Security, we also have attributes in the case of OAuth 2.0.

Let's say, for example, that you've got a `user_id` attribute that indicates the user's id in your system.

You might access it like so in a controller:

Java

```
@GetMapping("/endpoint")
public Mono<String> foo(BearerTokenAuthentication authentication) {
    String userId = (String) authentication.getTokenAttributes().get("user_id");
    // ...
}
```

Kotlin

```
@GetMapping("/endpoint")
fun foo(authentication: BearerTokenAuthentication): Mono<String?> {
    val userId = authentication.tokenAttributes["user_id"] as String?
    // ...
}
```

In that case, you'd want to specify that attribute with the `attributes()` method:

Java

```
client
    .mutateWith(mockOpaqueToken()
        .attributes(attrs -> attrs.put("user_id", "1234"))
    )
    .get().uri("/endpoint").exchange();
```

Kotlin

```
client
    .mutateWith(mockOpaqueToken()
        .attributes { attrs -> attrs["user_id"] = "1234" }
    )
    .get().uri("/endpoint").exchange()
```

Additional Configurations

There are additional methods, too, for further configuring the authentication; it simply depends on what data your controller expects.

One such is `principal(OAuth2AuthenticatedPrincipal)`, which you can use to configure the complete `OAuth2AuthenticatedPrincipal` instance that underlies the `BearerTokenAuthentication`

It's handy if you: 1. Have your own implementation of `OAuth2AuthenticatedPrincipal`, or 2. Want to specify a different principal name

For example, let's say that your authorization server sends the principal name in the `user_name` attribute instead of the `sub` attribute. In that case, you can configure an `OAuth2AuthenticatedPrincipal` by hand:

Java

```
Map<String, Object> attributes = Collections.singletonMap("user_name",
"foo_user");
OAuth2AuthenticatedPrincipal principal = new DefaultOAuth2AuthenticatedPrincipal(
        (String) attributes.get("user_name"),
        attributes,
        AuthorityUtils.createAuthorityList("SCOPE_message:read"));

client
    .mutateWith(mockOpaqueToken().principal(principal))
    .get().uri("/endpoint").exchange();
```

Kotlin

```
val attributes: Map<String, Any> = mapOf(Pair("user_name", "foo_user"))
val principal: OAuth2AuthenticatedPrincipal = DefaultOAuth2AuthenticatedPrincipal(
    attributes["user_name"] as String?,
    attributes,
    AuthorityUtils.createAuthorityList("SCOPE_message:read")
)

client
    .mutateWith(mockOpaqueToken().principal(principal))
    .get().uri("/endpoint").exchange()
```

Note that as an alternative to using `mockOpaqueToken()` test support, you can also mock the `OpaqueTokenIntrospector` bean itself with a `@MockBean` annotation.

Chapter 31. RSocket Security

Spring Security's RSocket support relies on a `SocketAcceptorInterceptor`. The main entry point into security is found in the `PayloadSocketAcceptorInterceptor` which adapts the RSocket APIs to allow intercepting a `PayloadExchange` with `PayloadInterceptor` implementations.

You can find a few sample applications that demonstrate the code below:

- Hello RSocket hellorsocket
- Spring Flights

31.1. Minimal RSocket Security Configuration

You can find a minimal RSocket Security configuration below:

Java

```
@Configuration
@EnableRSocketSecurity
public class HelloRSocketSecurityConfig {

    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
            .roles("USER")
            .build();
        return new MapReactiveUserDetailsService(user);
    }
}
```

Kotlin

```
@Configuration
@EnableRSocketSecurity
open class HelloRSocketSecurityConfig {
    @Bean
    open fun userDetailsService(): MapReactiveUserDetailsService {
        val user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
            .roles("USER")
            .build()
        return MapReactiveUserDetailsService(user)
    }
}
```

This configuration enables simple authentication and sets up rsocket-authorization to require an authenticated user for any request.

31.2. Adding SecuritySocketAcceptorInterceptor

For Spring Security to work we need to apply `SecuritySocketAcceptorInterceptor` to the `ServerRSocketFactory`. This is what connects our `PayloadSocketAcceptorInterceptor` we created with the RSocket infrastructure. In a Spring Boot application this is done automatically using `RSocketSecurityAutoConfiguration` with the following code.

```
@Bean
RSocketServerCustomizer
springSecurityRSocketSecurity(SecuritySocketAcceptorInterceptor interceptor) {
    return (server) -> server.interceptors((registry) ->
registry.forSocketAcceptor(interceptor));
}
```

31.3. RSocket Authentication

RSocket authentication is performed with `AuthenticationPayloadInterceptor` which acts as a controller to invoke a `ReactiveAuthenticationManager` instance.

31.3.1. Authentication at Setup vs Request Time

Generally, authentication can occur at setup time and/or request time.

Authentication at setup time makes sense in a few scenarios. A common scenarios is when a single user (i.e. mobile connection) is leveraging an RSocket connection. In this case only a single user is leveraging the connection, so authentication can be done once at connection time.

In a scenario where the RSocket connection is shared it makes sense to send credentials on each request. For example, a web application that connects to an RSocket server as a downstream service would make a single connection that all users leverage. In this case, if the RSocket server needs to perform authorization based on the web application's users credentials per request makes sense.

In some scenarios authentication at setup and per request makes sense. Consider a web application as described previously. If we need to restrict the connection to the web application itself, we can provide a credential with a `SETUP` authority at connection time. Then each user would have different authorities but not the `SETUP` authority. This means that individual users can make requests but not make additional connections.

31.3.2. Simple Authentication

Spring Security has support for Simple Authentication Metadata Extension.

Basic Authentication drafts evolved into Simple Authentication and is only supported for backward compatibility. See `RSocketSecurity.basicAuthentication(Customizer)` for setting it up.

The RSocket receiver can decode the credentials using `AuthenticationPayloadExchangeConverter` which is automatically setup using the `simpleAuthentication` portion of the DSL. An explicit configuration can be found below.

Java

```
@Bean
PayloadSocketAcceptorInterceptor rsocketInterceptor(RSocketSecurity rsocket) {
    rsocket
        .authorizePayload(authorize ->
            authorize
                    .anyRequest().authenticated()
                    .anyExchange().permitAll()
        )
        .simpleAuthentication(Customizer.withDefaults());
    return rsocket.build();
}
```

Kotlin

```
@Bean
open fun rsocketInterceptor(rsocket: RSocketSecurity):
PayloadSocketAcceptorInterceptor {
    rsocket
        .authorizePayload { authorize -> authorize
                .anyRequest().authenticated()
                .anyExchange().permitAll()
        }
        .simpleAuthentication(withDefaults())
    return rsocket.build()
}
```

The RSocket sender can send credentials using `SimpleAuthenticationEncoder` which can be added to Spring's `RSocketStrategies`.

Java

```
RSocketStrategies.Builder strategies = ...;
strategies.encoder(new SimpleAuthenticationEncoder());
```

Kotlin

```
var strategies: RSocketStrategies.Builder = ...
strategies.encoder(SimpleAuthenticationEncoder())
```

It can then be used to send a username and password to the receiver in the setup:

Java

```
MimeType authenticationMimeType =

MimeTypeUtils.parseMimeType(WellKnownMimeType.MESSAGE_RSOCKET_AUTHENTICATION.getSt
ring());
UsernamePasswordMetadata credentials = new UsernamePasswordMetadata("user",
"password");
Mono<RSocketRequester> requester = RSocketRequester.builder()
    .setupMetadata(credentials, authenticationMimeType)
    .rsocketStrategies(strategies.build())
    .connectTcp(host, port);
```

Kotlin

```
val authenticationMimeType: MimeType =

MimeTypeUtils.parseMimeType(WellKnownMimeType.MESSAGE_RSOCKET_AUTHENTICATION.strin
g)
val credentials = UsernamePasswordMetadata("user", "password")
val requester: Mono<RSocketRequester> = RSocketRequester.builder()
    .setupMetadata(credentials, authenticationMimeType)
    .rsocketStrategies(strategies.build())
    .connectTcp(host, port)
```

Alternatively or additionally, a username and password can be sent in a request.

Java

```
Mono<RSocketRequester> requester;
UsernamePasswordMetadata credentials = new UsernamePasswordMetadata("user",
"password");

public Mono<AirportLocation> findRadar(String code) {
    return this.requester.flatMap(req ->
        req.route("find.radar.{code}", code)
            .metadata(credentials, authenticationMimeType)
            .retrieveMono(AirportLocation.class)
    );
}
```

Kotlin

```
import org.springframework.messaging.rsocket.retrieveMono

// ...

var requester: Mono<RSocketRequester>? = null
var credentials = UsernamePasswordMetadata("user", "password")

open fun findRadar(code: String): Mono<AirportLocation> {
    return requester!!.flatMap { req ->
        req.route("find.radar.{code}", code)
            .metadata(credentials, authenticationMimeType)
            .retrieveMono<AirportLocation>()
    }
}
```

31.3.3. JWT

Spring Security has support for Bearer Token Authentication Metadata Extension. The support comes in the form of authenticating a JWT (determining the JWT is valid) and then using the JWT to make authorization decisions.

The RSocket receiver can decode the credentials using `BearerPayloadExchangeConverter` which is automatically setup using the `jwt` portion of the DSL. An example configuration can be found below:

Java

```
@Bean
PayloadSocketAcceptorInterceptor rsocketInterceptor(RSocketSecurity rsocket) {
    rsocket
        .authorizePayload(authorize ->
            authorize
                .anyRequest().authenticated()
                .anyExchange().permitAll()
        )
        .jwt(Customizer.withDefaults());
    return rsocket.build();
}
```

Kotlin

```
@Bean
fun rsocketInterceptor(rsocket: RSocketSecurity): PayloadSocketAcceptorInterceptor
{
    rsocket
        .authorizePayload { authorize -> authorize
            .anyRequest().authenticated()
            .anyExchange().permitAll()
        }
        .jwt(withDefaults())
    return rsocket.build()
}
```

The configuration above relies on the existence of a `ReactiveJwtDecoder @Bean` being present. An example of creating one from the issuer can be found below:

Java

```
@Bean
ReactiveJwtDecoder jwtDecoder() {
    return ReactiveJwtDecoders
        .fromIssuerLocation("https://example.com/auth/realms/demo");
}
```

Kotlin

```
@Bean
fun jwtDecoder(): ReactiveJwtDecoder {
    return ReactiveJwtDecoders
        .fromIssuerLocation("https://example.com/auth/realms/demo")
}
```

The RSocket sender does not need to do anything special to send the token because the value is just a simple String. For example, the token can be sent at setup time:

Java

```
MimeType authenticationMimeType =

MimeTypeUtils.parseMimeType(WellKnownMimeType.MESSAGE_RSOCKET_AUTHENTICATION.getSt
ring());
BearerTokenMetadata token = ...;
Mono<RSocketRequester> requester = RSocketRequester.builder()
    .setupMetadata(token, authenticationMimeType)
    .connectTcp(host, port);
```

Kotlin

```
val authenticationMimeType: MimeType =

MimeTypeUtils.parseMimeType(WellKnownMimeType.MESSAGE_RSOCKET_AUTHENTICATION.strin
g)
val token: BearerTokenMetadata = ...

val requester = RSocketRequester.builder()
    .setupMetadata(token, authenticationMimeType)
    .connectTcp(host, port)
```

Alternatively or additionally, the token can be sent in a request.

Java

```
MimeType authenticationMimeType =

MimeTypeUtils.parseMimeType(WellKnownMimeType.MESSAGE_RSOCKET_AUTHENTICATION.getSt
ring());
Mono<RSocketRequester> requester;
BearerTokenMetadata token = ...;

public Mono<AirportLocation> findRadar(String code) {
    return this.requester.flatMap(req ->
        req.route("find.radar.{code}", code)
            .metadata(token, authenticationMimeType)
            .retrieveMono(AirportLocation.class)
    );
}
```

Kotlin

```
val authenticationMimeType: MimeType =

MimeTypeUtils.parseMimeType(WellKnownMimeType.MESSAGE_RSOCKET_AUTHENTICATION.strin
g)
var requester: Mono<RSocketRequester>? = null
val token: BearerTokenMetadata = ...

open fun findRadar(code: String): Mono<AirportLocation> {
    return this.requester!!.flatMap { req ->
        req.route("find.radar.{code}", code)
            .metadata(token, authenticationMimeType)
            .retrieveMono<AirportLocation>()
    }
}
```

31.4. RSocket Authorization

RSocket authorization is performed with `AuthorizationPayloadInterceptor` which acts as a controller to invoke a `ReactiveAuthorizationManager` instance. The DSL can be used to setup authorization rules based upon the `PayloadExchange`. An example configuration can be found below:

Java

```
rsocket
    .authorizePayload(authz ->
        authz
            .setup().hasRole("SETUP") ①
            .route("fetch.profile.me").authenticated() ②
            .matcher(payloadExchange -> isMatch(payloadExchange)) ③
                .hasRole("CUSTOM")
            .route("fetch.profile.{username}") ④
                .access((authentication, context) -> checkFriends(authentication,
context))
            .anyRequest().authenticated() ⑤
            .anyExchange().permitAll() ⑥
    );
```

Kotlin

```
rsocket
    .authorizePayload { authz ->
        authz
            .setup().hasRole("SETUP") ①
            .route("fetch.profile.me").authenticated() ②
            .matcher { payloadExchange -> isMatch(payloadExchange) } ③
            .hasRole("CUSTOM")
            .route("fetch.profile.{username}") ④
            .access { authentication, context -> checkFriends(authentication,
context) }
            .anyRequest().authenticated() ⑤
            .anyExchange().permitAll()
    } ⑥
```

① Setting up a connection requires the authority `ROLE_SETUP`

② If the route is `fetch.profile.me` authorization only requires the user be authenticated

③ In this rule we setup a custom matcher where authorization requires the user to have the authority `ROLE_CUSTOM`

④ This rule leverages custom authorization. The matcher expresses a variable with the name `username` that is made available in the `context`. A custom authorization rule is exposed in the `checkFriends` method.

⑤ This rule ensures that request that does not already have a rule will require the user to be authenticated. A request is where the metadata is included. It would not include additional payloads.

⑥ This rule ensures that any exchange that does not already have a rule is allowed for anyone. In this example, it means that payloads that have no metadata have no authorization rules.

It is important to understand that authorization rules are performed in order. Only the first authorization rule that matches will be invoked.