Spring Session

Rob Winch, Vedran Pavić, Jay Bryant, Eleftheria Stein-Kousathana

Version 2.6.0-M1

Table of Contents

This documentation is also available as HTML.

Spring Session provides an API and implementations for managing a user's session information.

Chapter 1. Introduction

Spring Session provides an API and implementations for managing a user's session information while also making it trivial to support clustered sessions without being tied to an application container-specific solution. It also provides transparent integration with:

- HttpSession: Allows replacing the `HttpSession` in an application container-neutral way, with support for providing session IDs in headers to work with RESTful APIs.
- WebSocket: Provides the ability to keep the `HttpSession` alive when receiving WebSocket messages
- WebSession: Allows replacing the Spring WebFlux's `WebSession` in an application container-neutral way.

Chapter 2. What's New

Check also the Spring Session BOM release notes for a list of new and noteworthy features, as well as upgrade instructions for each release.

Chapter 3. Samples and Guides (Start Here)

To get started with Spring Session, the best place to start is our Sample Applications.

Table 1. Sample Applications that use Spring Boot

Source	Description	Guide
HttpSession with Redis	Demonstrates how to use Spring Session to replace the `HttpSession` with Redis.	HttpSession with Redis Guide
HttpSession with JDBC	Demonstrates how to use Spring Session to replace the `HttpSession` with a relational database store.	HttpSession with JDBC Guide
HttpSession with Hazelcast	Demonstrates how to use Spring Session to replace the `HttpSession` with Hazelcast.	
Find by Username	Demonstrates how to use Spring Session to find sessions by username.	Find by Username Guide
WebSockets	Demonstrates how to use Spring Session with WebSockets.	WebSockets Guide
WebFlux	Demonstrates how to use Spring Session to replace the Spring WebFlux's `WebSession` with Redis.	
WebFlux with Custom Cookie	Demonstrates how to use Spring Session to customize the Session cookie in a WebFlux based application.	WebFlux with Custom Cookie Guide
HttpSession with Redis JSON serialization	Demonstrates how to use Spring Session to replace the `HttpSession` with Redis using JSON serialization.	
HttpSession with simple Redis `SessionRepository`	Demonstrates how to use Spring Session to replace the `HttpSession` with Redis using `RedisSessionRepository`.	

Table 2. Sample Applications that use Spring Java-based configuration

Source	Description	Guide
HttpSession with Redis	Demonstrates how to use Spring Session to replace the `HttpSession` with Redis.	HttpSession with Redis Guide
HttpSession with JDBC	Demonstrates how to use Spring Session to replace the `HttpSession` with a relational database store.	HttpSession with JDBC Guide
HttpSession with Hazelcast	Demonstrates how to use Spring Session to replace the `HttpSession` with Hazelcast.	HttpSession with Hazelcast Guide
Custom Cookie	Demonstrates how to use Spring Session and customize the cookie.	Custom Cookie Guide
Spring Security	Demonstrates how to use Spring Session with an existing Spring Security application.	Spring Security Guide
REST	Demonstrates how to use Spring Session in a REST application to support authenticating with a header.	REST Guide

Table 3. Sample Applications that use Spring XML-based configuration

Source	Description	Guide
HttpSession with Redis	Demonstrates how to use Spring Session to replace the `HttpSession` with a Redis store.	HttpSession with Redis Guide
HttpSession with JDBC	Demonstrates how to use Spring Session to replace the `HttpSession` with a relational database store.	HttpSession with JDBC Guide

Table 4. Miscellaneous sample Applications

Source	Description	Guide
Hazelcast	Demonstrates how to use Spring Session with Hazelcast in a Java EE application.	

Chapter 4. Spring Session Modules

In Spring Session 1.x, all of the Spring Session's `SessionRepository` implementations were available within the `spring-session` artifact. While convenient, this approach was not sustainable long-term as more features and `SessionRepository` implementations were added to the project.

Starting with Spring Session 2.0, the project has been split into Spring Session Core module and several other modules that carry `SessionRepository` implementations and functionality related to the specific data store. Users of Spring Data should find this arrangement familiar, with Spring Session Core module taking a role equivalent to Spring Data Commons and providing core functionalities and APIs, with other modules containing data store specific implementations. As part of this split, the Spring Session Data MongoDB and Spring Session Data GemFire modules were moved to separate repositories. Now the situation with project's repositories/modules is as follows:

- `spring-session` repository
  - Hosts the Spring Session Core, Spring Session Data Redis, Spring Session JDBC, and Spring Session Hazelcast modules
- `spring-session-data-mongodb` repository
  - Hosts the Spring Session Data MongoDB module. Spring Session Data MongoDB has its own user guide, which you can find at the [https://spring.io/projects/spring-session-data-mongodb#learnSpring site].
- `spring-session-data-geode` repository
  - Hosts the Spring Session Data Geode modules. Spring Session Data Geode has its own user guide, which you can find at the [https://spring.io/projects/spring-session-data-geode#learn site].

Finally, Spring Session now also provides a Maven BOM ("bill of materials") module in order to help users with version management concerns:

- `spring-session-bom` repository
  - Hosts the Spring Session BOM module

Chapter 5. HttpSession Integration

Spring Session provides transparent integration with HttpSession. This means that developers can switch the HttpSession implementation out with an implementation that is backed by Spring Session.

5.1. Why Spring Session and HttpSession?

We have already mentioned that Spring Session provides transparent integration with HttpSession, but what benefits do we get out of this?

- **Clustered Sessions**: Spring Session makes it trivial to support clustered sessions without being tied to an application container specific solution.
- **RESTful APIs**: Spring Session lets providing session IDs in headers work with RESTful APIs

5.2. HttpSession with Redis

Using Spring Session with HttpSession is enabled by adding a Servlet Filter before anything that uses the HttpSession. You can choose from enabling this by using either:

- Java-based Configuration
- XML-based Configuration

5.2.1. Redis Java-based Configuration

This section describes how to use Redis to back HttpSession by using Java based configuration.

The HttpSession Sample provides a working sample of how to integrate Spring Session and HttpSession by using Java configuration. You can read the basic steps for integration in the next few sections, but we encourage you to follow along with the detailed HttpSession Guide when integrating with your own application.

Spring Java Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a servlet filter that replaces the HttpSession implementation with an implementation backed by Spring Session. To do so, add the following Spring Configuration:

```
@EnableRedisHttpSession ①
public class Config {

    @Bean
    public LettuceConnectionFactory connectionFactory() {
        return new LettuceConnectionFactory(); ②
    }

}
```

① The `@EnableRedisHttpSession` annotation creates a Spring Bean with the name of `springSessionRepositoryFilter` that implements `Filter`. The filter is in charge of replacing the `HttpSession` implementation to be backed by Spring Session. In this instance, Spring Session is backed by Redis.

② We create a `RedisConnectionFactory` that connects Spring Session to the Redis Server. We configure the connection to connect to localhost on the default port (6379). For more information on configuring Spring Data Redis, see the reference documentation.

Java Servlet Container Initialization

Our Spring Configuration created a Spring Bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, Spring needs to load our `Config` class. Last, we need to ensure that our Servlet Container (that is, Tomcat) uses our `springSessionRepositoryFilter` for every request. Fortunately, Spring Session provides a utility class named `AbstractHttpSessionApplicationInitializer` to make both of these steps easy. The following shows an example:

src/main/java/sample/Initializer.java

```
public class Initializer extends AbstractHttpSessionApplicationInitializer { ①

    public Initializer() {
        super(Config.class); ②
    }

}
```

The name of our class (`Initializer`) does not matter. What is important is that we extend `AbstractHttpSessionApplicationInitializer`.

① The first step is to extend `AbstractHttpSessionApplicationInitializer`. Doing so ensures that the Spring Bean by the name of `springSessionRepositoryFilter` is registered with our Servlet Container for every request.

② `AbstractHttpSessionApplicationInitializer` also provides a mechanism to ensure Spring loads our `Config`.

5.2.2. Redis XML-based Configuration

This section describes how to use Redis to back `HttpSession` by using XML based configuration.

The HttpSession XML Sample provides a working sample of how to integrate Spring Session and `HttpSession` using XML configuration. You can read the basic steps for integration in the next few sections, but we encourage you to follow along with the detailed HttpSession XML Guide when integrating with your own application.

Spring XML Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a servlet filter that replaces the `HttpSession` implementation with an implementation backed by Spring Session. To do so, add the following Spring Configuration:

src/main/webapp/WEB-INF/spring/session.xml

```
①
<context:annotation-config/>
<bean
class="org.springframework.session.data.redis.config.annotation.web.http.RedisHttp
SessionConfiguration"/>

②
<bean
class="org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory"
/>
```

① We use the combination of `<context:annotation-config/>` and `RedisHttpSessionConfiguration` because Spring Session does not yet provide XML Namespace support (see gh-104). This creates a Spring Bean with the name of `springSessionRepositoryFilter` that implements `Filter`. The filter is in charge of replacing the `HttpSession` implementation to be backed by Spring Session. In this instance, Spring Session is backed by Redis.

② We create a `RedisConnectionFactory` that connects Spring Session to the Redis Server. We configure the connection to connect to localhost on the default port (6379) For more information on configuring Spring Data Redis, see the reference documentation.

XML Servlet Container Initialization

Our Spring Configuration created a Spring Bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, we need to instruct Spring to load our `session.xml` configuration. We can do so with the following configuration:

src/main/webapp/WEB-INF/web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spring/session.xml
    </param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

The `ContextLoaderListener` reads the contextConfigLocation and picks up our session.xml configuration.

Last, we need to ensure that our Servlet Container (that is, Tomcat) uses our `springSessionRepositoryFilter` for every request. The following snippet performs this last step for us:

src/main/webapp/WEB-INF/web.xml

```
<filter>
    <filter-name>springSessionRepositoryFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
</filter>
<filter-mapping>
    <filter-name>springSessionRepositoryFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

The `DelegatingFilterProxy` looks up a Bean by the name of `springSessionRepositoryFilter` and cast it to a `Filter`. For every request that `DelegatingFilterProxy` is invoked, the `springSessionRepositoryFilter` is invoked.

5.3. `HttpSession` with JDBC

You can use Spring Session with `HttpSession` by adding a servlet filter before anything that uses the `HttpSession`. You can choose to do in any of the following ways:

- Java-based Configuration
- XML-based Configuration
- Spring Boot-based Configuration

5.3.1. JDBC Java-based Configuration

This section describes how to use a relational database to back `HttpSession` when you use Java-based configuration.

The HttpSession JDBC Sample provides a working sample of how to integrate Spring Session and `HttpSession` by using Java configuration. You can read the basic steps for integration in the next few sections, but we encouraged you to follow along with the detailed HttpSession JDBC Guide when integrating with your own application.

Spring Java Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a Servlet Filter that replaces the `HttpSession` implementation with an implementation backed by Spring Session. To do so, add the following Spring Configuration:

```
@EnableJdbcHttpSession ①
public class Config {

    @Bean
    public EmbeddedDatabase dataSource() {
        return new EmbeddedDatabaseBuilder() ②

.setType(EmbeddedDatabaseType.H2).addScript("org/springframework/session/jdbc/sche
ma-h2.sql").build();
    }

    @Bean
    public PlatformTransactionManager transactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource); ③
    }

}
```

① The `@EnableJdbcHttpSession` annotation creates a Spring Bean with the name of `springSessionRepositoryFilter`. That bean implements `Filter`. The filter is in charge of replacing the `HttpSession` implementation to be backed by Spring Session. In this instance, Spring Session is backed by a relational database.

② We create a `dataSource` that connects Spring Session to an embedded instance of an H2 database. We configure the H2 database to create database tables by using the SQL script that is included in Spring Session.

③ We create a `transactionManager` that manages transactions for previously configured `dataSource`.

For additional information on how to configure data access related concerns, see the Spring Framework Reference Documentation.

Java Servlet Container Initialization

Our Spring Configuration created a Spring bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, Spring needs to load our `Config` class. Last, we need to ensure that our Servlet Container (that is, Tomcat) uses our `springSessionRepositoryFilter` for every request. Fortunately, Spring Session provides a utility class named

`AbstractHttpSessionApplicationInitializer` to make both of these steps easy. The following example shows how to do so:

src/main/java/sample/Initializer.java

```
public class Initializer extends AbstractHttpSessionApplicationInitializer { ①

    public Initializer() {
        super(Config.class); ②
    }

}
```

The name of our class (Initializer) does not matter. What is important is that we extend `AbstractHttpSessionApplicationInitializer`.

① The first step is to extend `AbstractHttpSessionApplicationInitializer`. Doing so ensures that the Spring bean named `springSessionRepositoryFilter` is registered with our Servlet Container for every request.

② `AbstractHttpSessionApplicationInitializer` also provides a mechanism to ensure Spring loads our `Config`.

Multiple DataSources

Spring Session provides the `@SpringSessionDataSource` qualifier, allowing you to explicitly declare which `DataSource` bean should be injected in `JdbcIndexedSessionRepository`. This is particularly useful in scenarios with multiple `DataSource` beans present in the application context.

The following example shows how to do so:

Config.java

```
@EnableJdbcHttpSession
public class Config {

    @Bean
    @SpringSessionDataSource ①
    public EmbeddedDatabase firstDataSource() {
        return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.H2).addScript("org/springframework/session/jdbc/sche
ma-h2.sql").build();
    }

    @Bean
    public HikariDataSource secondDataSource() {
        // ...
    }
}
```

① This qualifier declares that firstDataSource is to be used by Spring Session.

5.3.2. JDBC XML-based Configuration

This section describes how to use a relational database to back `HttpSession` when you use XML based configuration.

The HttpSession JDBC XML Sample provides a working sample of how to integrate Spring Session and `HttpSession` by using XML configuration. You can read the basic steps for integration in the next few sections, but we encourage you to follow along with the detailed HttpSession JDBC XML Guide when integrating with your own application.

Spring XML Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a servlet filter that replaces the `HttpSession` implementation with an implementation backed by Spring Session. The following listing shows how to add the following Spring Configuration:

src/main/webapp/WEB-INF/spring/session.xml

```
①
<context:annotation-config/>
<bean
class="org.springframework.session.jdbc.config.annotation.web.http.JdbcHttpSession
Configuration"/>

②
<jdbc:embedded-database id="dataSource" database-name="testdb" type="H2">
    <jdbc:script location="classpath:org/springframework/session/jdbc/schema-
h2.sql"/>
</jdbc:embedded-database>

③
<bean class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <constructor-arg ref="dataSource"/>
</bean>
```

① We use the combination of `<context:annotation-config/>` and `JdbcHttpSessionConfiguration` because Spring Session does not yet provide XML Namespace support (see gh-104). This creates a Spring bean with the name of `springSessionRepositoryFilter`. That bean implements `Filter`. The filter is in charge of replacing the `HttpSession` implementation to be backed by Spring Session. In this instance, Spring Session is backed by a relational database.

② We create a `dataSource` that connects Spring Session to an embedded instance of an H2 database. We configure the H2 database to create database tables by using the SQL script that is included in Spring Session.

③ We create a `transactionManager` that manages transactions for previously configured `dataSource`.

For additional information on how to configure data access-related concerns, see the Spring Framework Reference Documentation.

XML Servlet Container Initialization

Our Spring Configuration created a Spring bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, we need to instruct Spring to load our `session.xml` configuration. We do so with the following configuration:

src/main/webapp/WEB-INF/web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spring/session.xml
    </param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

The `ContextLoaderListener` reads the `contextConfigLocation` and picks up our session.xml configuration.

Last, we need to ensure that our Servlet Container (that is, Tomcat) uses our `springSessionRepositoryFilter` for every request. The following snippet performs this last step for us:

src/main/webapp/WEB-INF/web.xml

```
<filter>
    <filter-name>springSessionRepositoryFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
</filter>
<filter-mapping>
    <filter-name>springSessionRepositoryFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

The `DelegatingFilterProxy` looks up a bean named `springSessionRepositoryFilter` and casts it to a `Filter`. For every request on which `DelegatingFilterProxy` is invoked, the `springSessionRepositoryFilter` is invoked.

5.3.3. JDBC Spring Boot-based Configuration

This section describes how to use a relational database to back `HttpSession` when you use Spring Boot.

The HttpSession JDBC Spring Boot Sample provides a working sample of how to integrate Spring Session and `HttpSession` by using Spring Boot. You can read the basic steps for integration in the next few sections, but we encourage you to follow along with the detailed HttpSession JDBC Spring Boot Guide when integrating with your own application.

Spring Boot Configuration

After adding the required dependencies, we can create our Spring Boot configuration. Thanks to first-class auto configuration support, setting up Spring Session backed by a relational database is as simple as adding a single configuration property to your `application.properties`. The following listing shows how to do so:

src/main/resources/application.properties

```
spring.session.store-type=jdbc # Session store type.
```

If a single Spring Session module is present on the classpath, Spring Boot uses that store implementation automatically. If you have more than one implementation, you must choose the StoreType that you wish to use to store the sessions, as shows above.

Under the hood, Spring Boot applies configuration that is equivalent to manually adding the `@EnableJdbcHttpSession` annotation. This creates a Spring bean with the name of `springSessionRepositoryFilter`. That bean implements `Filter`. The filter is in charge of replacing the `HttpSession` implementation to be backed by Spring Session.

You can further customize by using `application.properties`. The following listing shows how to do so:

src/main/resources/application.properties

```
server.servlet.session.timeout= # Session timeout. If a duration suffix is not
specified, seconds are used.
spring.session.jdbc.initialize-schema=embedded # Database schema initialization
mode.
spring.session.jdbc.schema=classpath:org/springframework/session/jdbc/schema-
@@platform@@.sql # Path to the SQL file to use to initialize the database schema.
spring.session.jdbc.table-name=SPRING_SESSION # Name of the database table used to
store sessions.
```

For more information, see the Spring Session portion of the Spring Boot documentation.

Configuring the `DataSource`

Spring Boot automatically creates a `DataSource` that connects Spring Session to an embedded instance of an H2 database. In a production environment, you need to update your configuration to

point to your relational database. For example, you can include the following in your application.properties:

src/main/resources/application.properties

```
spring.datasource.url= # JDBC URL of the database.
spring.datasource.username= # Login username of the database.
spring.datasource.password= # Login password of the database.
```

For more information, see the Configure a DataSource portion of the Spring Boot documentation.

Servlet Container Initialization

Our Spring Boot Configuration created a Spring bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, Spring needs to load our `Config` class. Last, we need to ensure that our Servlet Container (that is, Tomcat) uses our `springSessionRepositoryFilter` for every request. Fortunately, Spring Boot takes care of both of these steps for us.

5.4. HttpSession with Hazelcast

Using Spring Session with `HttpSession` is enabled by adding a Servlet Filter before anything that uses the `HttpSession`.

This section describes how to use Hazelcast to back `HttpSession` by using Java-based configuration.

The Hazelcast Spring Sample provides a working sample of how to integrate Spring Session and `HttpSession` by using Java configuration. You can read the basic steps for integration in the next few sections, but we encourage you to follow along with the detailed Hazelcast Spring Guide when integrating with your own application.

5.4.1. Spring Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a servlet filter that replaces the `HttpSession` implementation with an implementation backed by Spring Session. To do so, add the following Spring Configuration:

```
@EnableHazelcastHttpSession ①
@Configuration
public class HazelcastHttpSessionConfig {

    @Bean
    public HazelcastInstance hazelcastInstance() {
        Config config = new Config();
        MapAttributeConfig attributeConfig = new MapAttributeConfig()

.setName(HazelcastIndexedSessionRepository.PRINCIPAL_NAME_ATTRIBUTE)
                .setExtractor(PrincipalNameExtractor.class.getName());

config.getMapConfig(HazelcastIndexedSessionRepository.DEFAULT_SESSION_MAP_NAME) ②
                .addMapAttributeConfig(attributeConfig).addMapIndexConfig(
                        new
MapIndexConfig(HazelcastIndexedSessionRepository.PRINCIPAL_NAME_ATTRIBUTE,
false));
        SerializerConfig serializerConfig = new SerializerConfig();
        serializerConfig.setImplementation(new
HazelcastSessionSerializer()).setTypeClass(MapSession.class);
        config.getSerializationConfig().addSerializerConfig(serializerConfig); ③
        return Hazelcast.newHazelcastInstance(config); ④
    }

}
```

① The `@EnableHazelcastHttpSession` annotation creates a Spring bean named `springSessionRepositoryFilter` that implements `Filter`. The filter is in charge of replacing the `HttpSession` implementation to be backed by Spring Session. In this instance, Spring Session is backed by Hazelcast.

② In order to support retrieval of sessions by principal name index, an appropriate `ValueExtractor` needs to be registered. Spring Session provides `PrincipalNameExtractor` for this purpose.

③ In order to serialize `MapSession` objects efficiently, `HazelcastSessionSerializer` needs to be registered. If this is not set, Hazelcast will serialize sessions using native Java serialization.

④ We create a `HazelcastInstance` that connects Spring Session to Hazelcast. By default, the application starts and connects to an embedded instance of Hazelcast. For more information on configuring Hazelcast, see the reference documentation.

If `HazelcastSessionSerializer` is preferred, it needs to be configured for all Hazelcast cluster members before they start. In a Hazelcast cluster, all members should use the same serialization method for sessions. Also, if Hazelcast Client/Server topology is used, then both members and clients must use the same serialization method. The serializer can be registered via `ClientConfig` with the same `SerializerConfiguration` of members.

5.4.2. Servlet Container Initialization

Our Spring Configuration created a Spring bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, Spring needs to load our `SessionConfig` class. Since our application is already loading Spring configuration by using our `SecurityInitializer` class, we can add our `SessionConfig` class to it. The following listing shows how to do so:

src/main/java/sample/SecurityInitializer.java

```
public class SecurityInitializer extends AbstractSecurityWebApplicationInitializer
{

    public SecurityInitializer() {
        super(SecurityConfig.class, SessionConfig.class);
    }

}
```

Last, we need to ensure that our Servlet Container (that is, Tomcat) uses our `springSessionRepositoryFilter` for every request. It is extremely important that Spring Session's `springSessionRepositoryFilter` is invoked before Spring Security's `springSecurityFilterChain`. Doing so ensures that the `HttpSession` that Spring Security uses is backed by Spring Session. Fortunately, Spring Session provides a utility class named `AbstractHttpSessionApplicationInitializer` that makes this doing so easy. The following example shows how to do so:

src/main/java/sample/Initializer.java

```
public class Initializer extends AbstractHttpSessionApplicationInitializer {

}
```

The name of our class (`Initializer`) does not matter. What is important is that we extend `AbstractHttpSessionApplicationInitializer`.

By extending `AbstractHttpSessionApplicationInitializer`, we ensure that the Spring Bean named `springSessionRepositoryFilter` is registered with our servlet container for every request before Spring Security's `springSecurityFilterChain`.

5.5. How `HttpSession` Integration Works

Fortunately, both `HttpSession` and `HttpServletRequest` (the API for obtaining an `HttpSession`) are both interfaces. This means that we can provide our own implementations for each of these APIs.

This section describes how Spring Session provides transparent integration with `HttpSession`. We offer this content so that you can understand what is happening under the covers. This functionality is already integrated and you do NOT need to implement this logic yourself.

First, we create a custom `HttpServletRequest` that returns a custom implementation of `HttpSession`. It looks something like the following:

```
public class SessionRepositoryRequestWrapper extends HttpServletRequestWrapper {

    public SessionRepositoryRequestWrapper(HttpServletRequest original) {
        super(original);
    }

    public HttpSession getSession() {
        return getSession(true);
    }

    public HttpSession getSession(boolean createNew) {
        // create an HttpSession implementation from Spring Session
    }

    // ... other methods delegate to the original HttpServletRequest ...
}
```

Any method that returns an `HttpSession` is overridden. All other methods are implemented by `HttpServletRequestWrapper` and delegate to the original `HttpServletRequest` implementation.

We replace the `HttpServletRequest` implementation by using a servlet `Filter` called `SessionRepositoryFilter`. The following pseudocode shows how it works:

```
public class SessionRepositoryFilter implements Filter {

    public doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        SessionRepositoryRequestWrapper customRequest =
            new SessionRepositoryRequestWrapper(httpRequest);

        chain.doFilter(customRequest, response, chain);
    }

    // ...
}
```

By passing a custom `HttpServletRequest` implementation into the `FilterChain`, we ensure that anything invoked after our `Filter` uses the custom `HttpSession` implementation. This highlights why it is important that Spring Session's `SessionRepositoryFilter` be placed before anything that interacts with the `HttpSession`.

5.6. `HttpSession` and RESTful APIs

Spring Session can work with RESTful APIs by letting the session be provided in a header.

The REST Sample provides a working sample of how to use Spring Session in a REST application to support authenticating with a header. You can follow the basic steps for integration described in the next few sections, but we encourage you to follow along with the detailed REST Guide when integrating with your own application.

5.6.1. Spring Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a servlet filter that replaces the `HttpSession` implementation with an implementation backed by Spring Session. To do so, add the following Spring Configuration:

```
@Configuration
@EnableRedisHttpSession ①
public class HttpSessionConfig {

    @Bean
    public LettuceConnectionFactory connectionFactory() {
        return new LettuceConnectionFactory(); ②
    }

    @Bean
    public HttpSessionIdResolver httpSessionIdResolver() {
        return HeaderHttpSessionIdResolver.xAuthToken(); ③
    }

}
```

① The `@EnableRedisHttpSession` annotation creates a Spring bean named `springSessionRepositoryFilter` that implements `Filter`. The filter is in charge of replacing the `HttpSession` implementation to be backed by Spring Session. In this instance, Spring Session is backed by Redis.

② We create a `RedisConnectionFactory` that connects Spring Session to the Redis Server. We configure the connection to connect to localhost on the default port (6379). For more information on configuring Spring Data Redis, see the reference documentation.

③ We customize Spring Session's HttpSession integration to use HTTP headers to convey the current session information instead of cookies.

5.6.2. Servlet Container Initialization

Our Spring Configuration created a Spring Bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, Spring needs to load our `Config` class. We provide the configuration in our Spring `MvcInitializer`, as the following example shows:

src/main/java/sample/mvc/MvcInitializer.java

```
@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class[] { SecurityConfig.class, HttpSessionConfig.class };
}
```

Last, we need to ensure that our Servlet Container (that is, Tomcat) uses our `springSessionRepositoryFilter` for every request. Fortunately, Spring Session provides a utility class

named `AbstractHttpSessionApplicationInitializer` that makes doing so easy. To do so, extend the class with the default constructor, as the following example shows:

src/main/java/sample/Initializer.java

```
public class Initializer extends AbstractHttpSessionApplicationInitializer {

}
```

The name of our class (`Initializer`) does not matter. What is important is that we extend `AbstractHttpSessionApplicationInitializer`.

5.7. Using `HttpSessionListener`

Spring Session supports `HttpSessionListener` by translating `SessionDestroyedEvent` and `SessionCreatedEvent` into `HttpSessionEvent` by declaring `SessionEventHttpSessionListenerAdapter`. To use this support, you need to:

- Ensure your `SessionRepository` implementation supports and is configured to fire `SessionDestroyedEvent` and `SessionCreatedEvent`.
- Configure `SessionEventHttpSessionListenerAdapter` as a Spring bean.
- Inject every `HttpSessionListener` into the `SessionEventHttpSessionListenerAdapter`

If you use the configuration support documented in `HttpSession` with Redis, all you need to do is register every `HttpSessionListener` as a bean. For example, assume you want to support Spring Security's concurrency control and need to use `HttpSessionEventPublisher`. In that case, you can add `HttpSessionEventPublisher` as a bean. In Java configuration, this might look like the following:

```
@Configuration
@EnableRedisHttpSession
public class RedisHttpSessionConfig {

    @Bean
    public HttpSessionEventPublisher httpSessionEventPublisher() {
        return new HttpSessionEventPublisher();
    }

    // ...

}
```

In XML configuration, this might look like the following:

```
<bean class="org.springframework.security.web.session.HttpSessionEventPublisher"/>
```

Chapter 6. WebSocket Integration

Spring Session provides transparent integration with Spring's WebSocket support.

Spring Session's WebSocket support works only with Spring's WebSocket support. Specifically,it does not work with using JSR-356 directly, because JSR-356 does not have a mechanism for intercepting incoming WebSocket messages.

6.1. Why Spring Session and WebSockets?

So why do we need Spring Session when we use WebSockets?

Consider an email application that does much of its work through HTTP requests. However, there is also a chat application embedded within it that works over WebSocket APIs. If a user is actively chatting with someone, we should not timeout the `HttpSession`, since this would be a pretty poor user experience. However, this is exactly what JSR-356 does.

Another issue is that, according to JSR-356, if the `HttpSession` times out, any WebSocket that was created with that `HttpSession` and an authenticated user should be forcibly closed. This means that, if we are actively chatting in our application and are not using the HttpSession, we also do disconnect from our conversation.

6.2. WebSocket Usage

The WebSocket Sample provides a working sample of how to integrate Spring Session with WebSockets. You can follow the basic steps for integration described in the next few headings, but we encourage you to follow along with the detailed WebSocket Guide when integrating with your own application.

6.2.1. `HttpSession` Integration

Before using WebSocket integration, you should be sure that you have `HttpSession` Integration working first.

6.2.2. Spring Configuration

In a typical Spring WebSocket application, you would implement `WebSocketMessageBrokerConfigurer`. For example, the configuration might look something like the following:

```
@Configuration
@EnableScheduling
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/messages").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/queue/", "/topic/");
        registry.setApplicationDestinationPrefixes("/app");
    }

}
```

We can update our configuration to use Spring Session's WebSocket support. The following example shows how to do so:

src/main/java/samples/config/WebSocketConfig.java

```
@Configuration
@EnableScheduling
@EnableWebSocketMessageBroker
public class WebSocketConfig extends
AbstractSessionWebSocketMessageBrokerConfigurer<Session> { ①

    @Override
    protected void configureStompEndpoints(StompEndpointRegistry registry) { ②
        registry.addEndpoint("/messages").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/queue/", "/topic/");
        registry.setApplicationDestinationPrefixes("/app");
    }

}
```

To hook in the Spring Session support we only need to change two things:

① Instead of implementing `WebSocketMessageBrokerConfigurer`, we extend `AbstractSessionWebSocketMessageBrokerConfigurer`

② We rename the `registerStompEndpoints` method to `configureStompEndpoints`

What does `AbstractSessionWebSocketMessageBrokerConfigurer` do behind the scenes?

- `WebSocketConnectHandlerDecoratorFactory` is added as a `WebSocketHandlerDecoratorFactory` to `WebSocketTransportRegistration`. This ensures a custom `SessionConnectEvent` is fired that contains the `WebSocketSession`. The `WebSocketSession` is necessary to end any WebSocket connections that are still open when a Spring Session is ended.
- `SessionRepositoryMessageInterceptor` is added as a `HandshakeInterceptor` to every `StompWebSocketEndpointRegistration`. This ensures that the `Session` is added to the WebSocket properties to enable updating the last accessed time.
- `SessionRepositoryMessageInterceptor` is added as a `ChannelInterceptor` to our inbound `ChannelRegistration`. This ensures that every time an inbound message is received, that the last accessed time of our Spring Session is updated.
- `WebSocketRegistryListener` is created as a Spring bean. This ensures that we have a mapping of all of the `Session` IDs to the corresponding WebSocket connections. By maintaining this mapping, we can close all the WebSocket connections when a Spring Session (HttpSession) is ended.

Chapter 7. WebSession Integration

Spring Session provides transparent integration with Spring WebFlux's `WebSession`. This means that you can switch the `WebSession` implementation out with an implementation that is backed by Spring Session.

7.1. Why Spring Session and WebSession?

We have already mentioned that Spring Session provides transparent integration with Spring WebFlux's `WebSession`, but what benefits do we get out of this? As with `HttpSession`, Spring Session makes it trivial to support clustered sessions without being tied to an application container specific solution.

7.2. WebSession with Redis

Using Spring Session with `WebSession` is enabled by registering a `WebSessionManager` implementation backed by Spring Session's `ReactiveSessionRepository`. The Spring configuration is responsible for creating a `WebSessionManager` that replaces the `WebSession` implementation with an implementation backed by Spring Session. To do so, add the following Spring Configuration:

```
@EnableRedisWebSession ①
public class SessionConfiguration {

    @Bean
    public LettuceConnectionFactory redisConnectionFactory() {
        return new LettuceConnectionFactory(); ②
    }

}
```

① The `@EnableRedisWebSession` annotation creates a Spring bean with the name of `webSessionManager`. That bean implements the `WebSessionManager`. This is what is in charge of replacing the `WebSession` implementation to be backed by Spring Session. In this instance, Spring Session is backed by Redis.

② We create a `RedisConnectionFactory` that connects Spring Session to the Redis Server. We configure the connection to connect to localhost on the default port (6379) For more information on configuring Spring Data Redis, see the reference documentation.

7.3. How WebSession Integration Works

It is considerably easier for Spring Session to integrate with Spring WebFlux and its `WebSession`, compared to Servlet API and its `HttpSession`. Spring WebFlux provides the `WebSessionStore` API, which presents a strategy for persisting `WebSession`.

This section describes how Spring Session provides transparent integration with `WebSession`. We offer this content so that you can understand what is happening under the covers. This functionality is already integrated and you do NOT need to implement this logic yourself.

First, we create a custom `SpringSessionWebSession` that delegates to Spring Session's `Session`. It looks something like the following:

```
public class SpringSessionWebSession implements WebSession {

    enum State {
        NEW, STARTED
    }

    private final S session;

    private AtomicReference<State> state = new AtomicReference<>();

    SpringSessionWebSession(S session, State state) {
        this.session = session;
        this.state.set(state);
    }

    @Override
    public void start() {
        this.state.compareAndSet(State.NEW, State.STARTED);
    }

    @Override
    public boolean isStarted() {
        State value = this.state.get();
        return (State.STARTED.equals(value)
                || (State.NEW.equals(value) &&
!this.session.getAttributes().isEmpty()));
    }

    @Override
    public Mono<Void> changeSessionId() {
        return Mono.defer(() -> {
            this.session.changeSessionId();
            return save();
        });
    }

    // ... other methods delegate to the original Session
}
```

Next, we create a custom `WebSessionStore` that delegates to the `ReactiveSessionRepository` and wraps `Session` into custom `WebSession` implementation, as the following listing shows:

```
public class SpringSessionWebSessionStore<S extends Session> implements
WebSessionStore {

    private final ReactiveSessionRepository<S> sessions;

    public SpringSessionWebSessionStore(ReactiveSessionRepository<S>
reactiveSessionRepository) {
        this.sessions = reactiveSessionRepository;
    }

    // ...
}
```

To be detected by Spring WebFlux, this custom `WebSessionStore` needs to be registered with `ApplicationContext` as a bean named `webSessionManager`. For additional information on Spring WebFlux, see the Spring Framework Reference Documentation.

Chapter 8. Spring Security Integration

Spring Session provides integration with Spring Security.

8.1. Spring Security Remember-me Support

Spring Session provides integration with Spring Security's Remember-me Authentication. The support:

- Changes the session expiration length
- Ensures that the session cookie expires at `Integer.MAX_VALUE`. The cookie expiration is set to the largest possible value, because the cookie is set only when the session is created. If it were set to the same value as the session expiration, the session would get renewed when the user used it but the cookie expiration would not be updated (causing the expiration to be fixed).

To configure Spring Session with Spring Security in Java Configuration, you can use the following listing as a guide:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        // ... additional configuration ...
        .rememberMe((rememberMe) -> rememberMe
            .rememberMeServices(rememberMeServices())
        );
}

@Bean
public SpringSessionRememberMeServices rememberMeServices() {
    SpringSessionRememberMeServices rememberMeServices =
            new SpringSessionRememberMeServices();
    // optionally customize
    rememberMeServices.setAlwaysRemember(true);
    return rememberMeServices;
}
```

An XML-based configuration would look something like the following:

```
<security:http>
    <!-- ... -->
    <security:form-login />
    <security:remember-me services-ref="rememberMeServices"/>
</security:http>

<bean id="rememberMeServices"

class="org.springframework.session.security.web.authentication.SpringSessionRememb
erMeServices"
    p:alwaysRemember="true"/>
```

8.2. Spring Security Concurrent Session Control

Spring Session provides integration with Spring Security to support its concurrent session control. This allows limiting the number of active sessions that a single user can have concurrently, but, unlike the default Spring Security support, this also works in a clustered environment. This is done by providing a custom implementation of Spring Security's `SessionRegistry` interface.

When using Spring Security's Java config DSL, you can configure the custom `SessionRegistry` through the `SessionManagementConfigurer`, as the following listing shows:

```
@Configuration
public class SecurityConfiguration<S extends Session> extends
WebSecurityConfigurerAdapter {

    @Autowired
    private FindByIndexNameSessionRepository<S> sessionRepository;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // @formatter:off
        http
            // other config goes here...
            .sessionManagement((sessionManagement) -> sessionManagement
                .maximumSessions(2)
                .sessionRegistry(sessionRegistry())
            );
        // @formatter:on
    }

    @Bean
    public SpringSessionBackedSessionRegistry<S> sessionRegistry() {
        return new SpringSessionBackedSessionRegistry<>(this.sessionRepository);
    }

}
```

This assumes that you have also configured Spring Session to provide a `FindByIndexNameSessionRepository` that returns `Session` instances.

When using XML configuration, it would look something like the following listing:

```
<security:http>
    <!-- other config goes here... -->
    <security:session-management>
        <security:concurrency-control max-sessions="2" session-registry-
ref="sessionRegistry"/>
    </security:session-management>
</security:http>

<bean id="sessionRegistry"

class="org.springframework.session.security.SpringSessionBackedSessionRegistry">
    <constructor-arg ref="sessionRepository"/>
</bean>
```

This assumes that your Spring Session `SessionRegistry` bean is called `sessionRegistry`, which is the name used by all `SpringHttpSessionConfiguration` subclasses.

8.3. Limitations

Spring Session's implementation of Spring Security's `SessionRegistry` interface does not support the `getAllPrincipals` method, as this information cannot be retrieved by using Spring Session. This method is never called by Spring Security, so this affects only applications that access the `SessionRegistry` themselves.

Chapter 9. API Documentation

You can browse the complete Javadoc online. The key APIs are described in the following sections:

- Using `Session`
- Using `SessionRepository`
- Using `FindByIndexNameSessionRepository`
- Using `ReactiveSessionRepository`
- Using `@EnableSpringHttpSession`
- Using `@EnableSpringWebSession`
- Using `RedisIndexedSessionRepository`
- Using `ReactiveRedisSessionRepository`
- Using `MapSessionRepository`
- Using `ReactiveMapSessionRepository`
- Using `JdbcIndexedSessionRepository`
- Using `HazelcastIndexedSessionRepository`
- Using `CookieSerializer`

9.1. Using `Session`

A `Session` is a simplified `Map` of name value pairs.

Typical usage might look like the following listing:

```
public class RepositoryDemo<S extends Session> {

    private SessionRepository<S> repository; ①

    public void demo() {
        S toSave = this.repository.createSession(); ②

        ③
        User rwinch = new User("rwinch");
        toSave.setAttribute(ATTR_USER, rwinch);

        this.repository.save(toSave); ④

        S session = this.repository.findById(toSave.getId()); ⑤

        ⑥
        User user = session.getAttribute(ATTR_USER);
        assertThat(user).isEqualTo(rwinch);
    }

    // ... setter methods ...

}
```

① We create a `SessionRepository` instance with a generic type, `S`, that extends `Session`. The generic type is defined in our class.

② We create a new `Session` by using our `SessionRepository` and assign it to a variable of type `S`.

③ We interact with the `Session`. In our example, we demonstrate saving a `User` to the `Session`.

④ We now save the `Session`. This is why we needed the generic type `S`. The `SessionRepository` only allows saving `Session` instances that were created or retrieved by using the same `SessionRepository`. This allows for the `SessionRepository` to make implementation specific optimizations (that is, writing only attributes that have changed).

⑤ We retrieve the `Session` from the `SessionRepository`.

⑥ We obtain the persisted `User` from our `Session` without the need for explicitly casting our attribute.

The `Session` API also provides attributes related to the `Session` instance's expiration.

Typical usage might look like the following listing:

```
public class ExpiringRepositoryDemo<S extends Session> {

    private SessionRepository<S> repository; ①

    public void demo() {
        S toSave = this.repository.createSession(); ②
        // ...
        toSave.setMaxInactiveInterval(Duration.ofSeconds(30)); ③

        this.repository.save(toSave); ④

        S session = this.repository.findById(toSave.getId()); ⑤
        // ...
    }

    // ... setter methods ...

}
```

① We create a `SessionRepository` instance with a generic type, `S`, that extends `Session`. The generic type is defined in our class.

② We create a new `Session` by using our `SessionRepository` and assign it to a variable of type `S`.

③ We interact with the `Session`. In our example, we demonstrate updating the amount of time the `Session` can be inactive before it expires.

④ We now save the `Session`. This is why we needed the generic type, `S`. The `SessionRepository` allows saving only `Session` instances that were created or retrieved using the same `SessionRepository`. This allows for the `SessionRepository` to make implementation specific optimizations (that is, writing only attributes that have changed). The last accessed time is automatically updated when the `Session` is saved.

⑤ We retrieve the `Session` from the `SessionRepository`. If the `Session` were expired, the result would be null.

9.2. Using `SessionRepository`

A `SessionRepository` is in charge of creating, retrieving, and persisting `Session` instances.

If possible, you should not interact directly with a `SessionRepository` or a `Session`. Instead, developers should prefer interacting with `SessionRepository` and `Session` indirectly through the `HttpSession` and WebSocket integration.

9.3. Using `FindByIndexNameSessionRepository`

Spring Session's most basic API for using a `Session` is the `SessionRepository`. This API is intentionally very simple, so that you can easily provide additional implementations with basic functionality.

Some `SessionRepository` implementations may also choose to implement `FindByIndexNameSessionRepository`. For example, Spring's Redis, JDBC, and Hazelcast support libraries all implement `FindByIndexNameSessionRepository`.

The `FindByIndexNameSessionRepository` provides a method to look up all the sessions with a given index name and index value. As a common use case that is supported by all provided `FindByIndexNameSessionRepository` implementations, you can use a convenient method to look up all the sessions for a particular user. This is done by ensuring that the session attribute with the name of `FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME` is populated with the username. It is your responsibility to ensure that the attribute is populated, since Spring Session is not aware of the authentication mechanism being used. An example of how to use this can be seen in the following listing:

```
String username = "username";
this.session.setAttribute(FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NA
ME, username);
```

Some implementations of `FindByIndexNameSessionRepository` provide hooks to automatically index other session attributes. For example, many implementations automatically ensure that the current Spring Security user name is indexed with the index name of `FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME`.

Once the session is indexed, you can find by using code similar to the following:

```
String username = "username";
Map<String, Session> sessionIdToSession =
this.sessionRepository.findByPrincipalName(username);
```

9.4. Using `ReactiveSessionRepository`

A `ReactiveSessionRepository` is in charge of creating, retrieving, and persisting `Session` instances in a non-blocking and reactive manner.

If possible, you should not interact directly with a `ReactiveSessionRepository` or a `Session`. Instead, you should prefer interacting with `ReactiveSessionRepository` and `Session` indirectly through the WebSession integration.

9.5. Using `@EnableSpringHttpSession`

You can add the `@EnableSpringHttpSession` annotation to a `@Configuration` class to expose the `SessionRepositoryFilter` as a bean named `springSessionRepositoryFilter`. In order to use the annotation, you must provide a single `SessionRepository` bean. The following example shows how to do so:

```
@EnableSpringHttpSession
@Configuration
public class SpringHttpSessionConfig {

    @Bean
    public MapSessionRepository sessionRepository() {
        return new MapSessionRepository(new ConcurrentHashMap<>());
    }

}
```

Note that no infrastructure for session expirations is configured for you. This is because things such as session expiration are highly implementation-dependent. This means that, if you need to clean up expired sessions, you are responsible for cleaning up the expired sessions.

9.6. Using `@EnableSpringWebSession`

You can add the `@EnableSpringWebSession` annotation to a `@Configuration` class to expose the `WebSessionManager` as a bean named `webSessionManager`. To use the annotation, you must provide a single `ReactiveSessionRepository` bean. The following example shows how to do so:

```
@EnableSpringWebSession
public class SpringWebSessionConfig {

    @Bean
    public ReactiveSessionRepository reactiveSessionRepository() {
        return new ReactiveMapSessionRepository(new ConcurrentHashMap<>());
    }

}
```

Note that no infrastructure for session expirations is configured for you. This is because things such as session expiration are highly implementation-dependent. This means that, if you require cleaning up expired sessions, you are responsible for cleaning up the expired sessions.

9.7. Using `RedisIndexedSessionRepository`

`RedisIndexedSessionRepository` is a `SessionRepository` that is implemented by using Spring Data's `RedisOperations`. In a web environment, this is typically used in combination with `SessionRepositoryFilter`. The implementation supports `SessionDestroyedEvent` and `SessionCreatedEvent` through `SessionMessageListener`.

9.7.1. Instantiating a `RedisIndexedSessionRepository`

You can see a typical example of how to create a new instance in the following listing:

```
RedisTemplate<Object, Object> redisTemplate = new RedisTemplate<>();

// ... configure redisTemplate ...

SessionRepository<? extends Session> repository = new
RedisIndexedSessionRepository(redisTemplate);
```

For additional information on how to create a `RedisConnectionFactory`, see the Spring Data Redis Reference.

9.7.2. Using `@EnableRedisHttpSession`

In a web environment, the simplest way to create a new `RedisIndexedSessionRepository` is to use `@EnableRedisHttpSession`. You can find complete example usage in the Samples and Guides (Start Here). You can use the following attributes to customize the configuration:

- **maxInactiveIntervalInSeconds**: The amount of time before the session expires, in seconds.
- **redisNamespace**: Allows configuring an application specific namespace for the sessions. Redis keys and channel IDs start with the prefix of `<redisNamespace>:`.
- **flushMode**: Allows specifying when data is written to Redis. The default is only when `save` is invoked on `SessionRepository`. A value of `FlushMode.IMMEDIATE` writes to Redis as soon as possible.

Custom `RedisSerializer`

You can customize the serialization by creating a bean named `springSessionDefaultRedisSerializer` that implements `RedisSerializer<Object>`.

9.7.3. Redis `TaskExecutor`

`RedisIndexedSessionRepository` is subscribed to receive events from Redis by using a `RedisMessageListenerContainer`. You can customize the way those events are dispatched by creating a bean named `springSessionRedisTaskExecutor`, a bean `springSessionRedisSubscriptionExecutor`, or both. You can find more details on configuring Redis task executors here.

9.7.4. Storage Details

The following sections outline how Redis is updated for each operation. The following example shows an example of creating a new session:

```
HMSET spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe creationTime
1404360000000 \
    maxInactiveInterval 1800 \
    lastAccessedTime 1404360000000 \
    sessionAttr:attrName someAttrValue \
    sessionAttr:attrName2 someAttrValue2
EXPIRE spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe 2100
APPEND spring:session:sessions:expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe ""
EXPIRE spring:session:sessions:expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe 1800
SADD spring:session:expirations:1439245080000 expires:33fdd1b6-b496-4b33-9f7d-
df96679d32fe
EXPIRE spring:session:expirations1439245080000 2100
```

The subsequent sections describe the details.

Saving a Session

Each session is stored in Redis as a `Hash`. Each session is set and updated by using the `HMSET` command. The following example shows how each session is stored:

```
HMSET spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe creationTime
1404360000000 \
    maxInactiveInterval 1800 \
    lastAccessedTime 1404360000000 \
    sessionAttr:attrName someAttrValue \
    sessionAttr:attrName2 someAttrValue2
```

In the preceding example, the following statements are true about the session:

- The session ID is 33fdd1b6-b496-4b33-9f7d-df96679d32fe.
- The session was created at 1404360000000 (in milliseconds since midnight of 1/1/1970 GMT).
- The session expires in 1800 seconds (30 minutes).
- The session was last accessed at 1404360000000 (in milliseconds since midnight of 1/1/1970 GMT).
- The session has two attributes. The first is `attrName`, with a value of `someAttrValue`. The second session attribute is named `attrName2`, with a value of `someAttrValue2`.

Optimized Writes

The `Session` instances managed by `RedisIndexedSessionRepository` keeps track of the properties that have changed and updates only those. This means that, if an attribute is written once and read many times, we need to write that attribute only once. For example, assume the `attrName2` session attribute from the lsiting in the preceding section was updated. The following command would be

run upon saving:

```
HMSET spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe
sessionAttr:attrName2 newValue
```

Session Expiration

An expiration is associated with each session by using the `EXPIRE` command, based upon the `Session.getMaxInactiveInterval()`. The following example shows a typical `EXPIRE` command:

```
EXPIRE spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe 2100
```

Note that the expiration that is set to five minutes after the session actually expires. This is necessary so that the value of the session can be accessed when the session expires. An expiration is set on the session itself five minutes after it actually expires to ensure that it is cleaned up, but only after we perform any necessary processing.

The `SessionRepository.findById(String)` method ensures that no expired sessions are returned. This means that you need not check the expiration before using a session.

Spring Session relies on the delete and expired keyspace notifications from Redis to fire a `SessionDeletedEvent` and a `SessionExpiredEvent`, respectively. `SessionDeletedEvent` or `SessionExpiredEvent` ensure that resources associated with the `Session` are cleaned up. For example, when you use Spring Session's WebSocket support, the Redis expired or delete event triggers any WebSocket connections associated with the session to be closed.

Expiration is not tracked directly on the session key itself, since this would mean the session data would no longer be available. Instead, a special session expires key is used. In the preceding example, the expires key is as follows:

```
APPEND spring:session:sessions:expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe ""
EXPIRE spring:session:sessions:expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe 1800
```

When a session expires key is deleted or expires, the keyspace notification triggers a lookup of the actual session, and a `SessionDestroyedEvent` is fired.

One problem with relying on Redis expiration exclusively is that, if the key has not been accessed, Redis makes no guarantee of when the expired event is fired. Specifically, the background task that Redis uses to clean up expired keys is a low-priority task and may not trigger the key expiration. For additional details, see the Timing of Expired Events section in the Redis documentation.

To circumvent the fact that expired events are not guaranteed to happen, we can ensure that each key is accessed when it is expected to expire. This means that, if the TTL is expired on the key, Redis removes the key and fires the expired event when we try to access the key.

For this reason, each session expiration is also tracked to the nearest minute. This lets a background task access the potentially expired sessions to ensure that Redis expired events are fired in a more deterministic fashion. The following example shows these events:

```
SADD spring:session:expirations:1439245080000 expires:33fdd1b6-b496-4b33-9f7d-
df96679d32fe
EXPIRE spring:session:expirations1439245080000 2100
```

The background task then uses these mappings to explicitly request each key. By accessing the key, rather than deleting it, we ensure that Redis deletes the key for us only if the TTL is expired.

We do not explicitly delete the keys, since, in some instances, there may be a race condition that incorrectly identifies a key as expired when it is not. Short of using distributed locks (which would kill our performance), there is no way to ensure the consistency of the expiration mapping. By simply accessing the key, we ensure that the key is only removed if the TTL on that key is expired.

9.7.5. `SessionDeletedEvent` and `SessionExpiredEvent`

`SessionDeletedEvent` and `SessionExpiredEvent` are both types of `SessionDestroyedEvent`.

`RedisIndexedSessionRepository` supports firing a `SessionDeletedEvent` when a `Session` is deleted or a `SessionExpiredEvent` when a `Session` expires. This is necessary to ensure resources associated with the `Session` are properly cleaned up.

For example, when integrating with WebSockets, the `SessionDestroyedEvent` is in charge of closing any active WebSocket connections.

Firing `SessionDeletedEvent` or `SessionExpiredEvent` is made available through the `SessionMessageListener`, which listens to Redis Keyspace events. In order for this to work, Redis Keyspace events for Generic commands and Expired events needs to be enabled. The following example shows how to do so:

```
redis-cli config set notify-keyspace-events Egx
```

If you use `@EnableRedisHttpSession`, managing the `SessionMessageListener` and enabling the necessary Redis Keyspace events is done automatically. However, in a secured Redis enviornment, the config command is disabled. This means that Spring Session cannot configure Redis Keyspace events for you. To disable the automatic configuration, add `ConfigureRedisAction.NO_OP` as a bean.

For example, with Java configuration, you can use the following:

```
@Bean
ConfigureRedisAction configureRedisAction() {
    return ConfigureRedisAction.NO_OP;
}
```

In XML configuration, you can use the following:

```
<util:constant
    static-
field="org.springframework.session.data.redis.config.ConfigureRedisAction.NO_OP"/>
```

9.7.6. Using `SessionCreatedEvent`

When a session is created, an event is sent to Redis with a channel ID of `spring:session:channel:created:33fdd1b6-b496-4b33-9f7d-df96679d32fe`, where `33fdd1b6-b496-4b33-9f7d-df96679d32fe` is the session ID. The body of the event is the session that was created.

If registered as a `MessageListener` (the default), `RedisIndexedSessionRepository` then translates the Redis message into a `SessionCreatedEvent`.

9.7.7. Viewing the Session in Redis

After installing redis-cli, you can inspect the values in Redis using the redis-cli. For example, you can enter the following into a terminal:

```
$ redis-cli
redis 127.0.0.1:6379> keys *
1) "spring:session:sessions:4fc39ce3-63b3-4e17-b1c4-5e1ed96fb021" ①
2) "spring:session:expirations:1418772300000" ②
```

① The suffix of this key is the session identifier of the Spring Session.

② This key contains all the session IDs that should be deleted at the time `1418772300000`.

You can also view the attributes of each session. The following example shows how to do so:

```
redis 127.0.0.1:6379> hkeys spring:session:sessions:4fc39ce3-63b3-4e17-b1c4-
5e1ed96fb021
1) "lastAccessedTime"
2) "creationTime"
3) "maxInactiveInterval"
4) "sessionAttr:username"
redis 127.0.0.1:6379> hget spring:session:sessions:4fc39ce3-63b3-4e17-b1c4-
5e1ed96fb021 sessionAttr:username
"\xac\xed\x00\x05t\x00\x03rob"
```

9.8. Using `ReactiveRedisSessionRepository`

`ReactiveRedisSessionRepository` is a `ReactiveSessionRepository` that is implemented by using Spring Data's `ReactiveRedisOperations`. In a web environment, this is typically used in combination with `WebSessionStore`.

9.8.1. Instantiating a `ReactiveRedisSessionRepository`

The following example shows how to create a new instance:

```
// ... create and configure connectionFactory and serializationContext ...

ReactiveRedisTemplate<String, Object> redisTemplate = new
ReactiveRedisTemplate<>(connectionFactory,
        serializationContext);

ReactiveSessionRepository<? extends Session> repository = new
ReactiveRedisSessionRepository(redisTemplate);
```

For additional information on how to create a `ReactiveRedisConnectionFactory`, see the Spring Data Redis Reference.

9.8.2. Using `@EnableRedisWebSession`

In a web environment, the simplest way to create a new `ReactiveRedisSessionRepository` is to use `@EnableRedisWebSession`. You can use the following attributes to customize the configuration:

- **maxInactiveIntervalInSeconds**: The amount of time before the session expires, in seconds
- **redisNamespace**: Allows configuring an application specific namespace for the sessions. Redis keys and channel IDs start with q prefix of `<redisNamespace>:`.
- **flushMode**: Allows specifying when data is written to Redis. The default is only when `save` is invoked on `ReactiveSessionRepository`. A value of `FlushMode.IMMEDIATE` writes to Redis as soon as possible.

Optimized Writes

The `Session` instances managed by `ReactiveRedisSessionRepository` keep track of the properties that have changed and updates only those. This means that, if an attribute is written once and read many times, we need to write that attribute only once.

9.8.3. Viewing the Session in Redis

After installing redis-cli, you can inspect the values in Redis using the redis-cli. For example, you can enter the following command into a terminal window:

```
$ redis-cli
redis 127.0.0.1:6379> keys *
1) "spring:session:sessions:4fc39ce3-63b3-4e17-b1c4-5e1ed96fb021" ①
```

① The suffix of this key is the session identifier of the Spring Session.

You can also view the attributes of each session by using the `hkeys` command. The following example shows how to do so:

```
redis 127.0.0.1:6379> hkeys spring:session:sessions:4fc39ce3-63b3-4e17-b1c4-
5e1ed96fb021
1) "lastAccessedTime"
2) "creationTime"
3) "maxInactiveInterval"
4) "sessionAttr:username"
redis 127.0.0.1:6379> hget spring:session:sessions:4fc39ce3-63b3-4e17-b1c4-
5e1ed96fb021 sessionAttr:username
"\xac\xed\x00\x05t\x00\x03rob"
```

9.9. Using `MapSessionRepository`

The `MapSessionRepository` allows for persisting `Session` in a `Map`, with the key being the `Session` ID and the value being the `Session`. You can use the implementation with a `ConcurrentHashMap` as a testing or convenience mechanism. Alternatively, you can use it with distributed `Map` implementations. For example, it can be used with Hazelcast.

9.9.1. Instantiating `MapSessionRepository`

The following example shows how to create a new instance:

```
SessionRepository<? extends Session> repository = new MapSessionRepository(new
ConcurrentHashMap<>());
```

9.9.2. Using Spring Session and Hazlecast

The Hazelcast Sample is a complete application that demonstrates how to use Spring Session with Hazelcast.

To run it, use the following command:

```
    ./gradlew :samples:hazelcast:tomcatRun
```

The Hazelcast Spring Sample is a complete application that demonstrates how to use Spring Session with Hazelcast and Spring Security.

It includes example Hazelcast `MapListener` implementations that support firing `SessionCreatedEvent`, `SessionDeletedEvent`, and `SessionExpiredEvent`.

To run it, use the following command:

```
    ./gradlew :samples:hazelcast-spring:tomcatRun
```

9.10. Using `ReactiveMapSessionRepository`

The `ReactiveMapSessionRepository` allows for persisting `Session` in a `Map`, with the key being the `Session` ID and the value being the `Session`. You can use the implementation with a `ConcurrentHashMap` as a testing or convenience mechanism. Alternatively, you can use it with distributed `Map` implementations, with the requirement that the supplied `Map` must be non-blocking.

9.11. Using `JdbcIndexedSessionRepository`

`JdbcIndexedSessionRepository` is a `SessionRepository` implementation that uses Spring's `JdbcOperations` to store sessions in a relational database. In a web environment, this is typically used in combination with `SessionRepositoryFilter`. Note that this implementation does not support publishing of session events.

9.11.1. Instantiating a `JdbcIndexedSessionRepository`

The following example shows how to create a new instance:

```
JdbcTemplate jdbcTemplate = new JdbcTemplate();

// ... configure jdbcTemplate ...

TransactionTemplate transactionTemplate = new TransactionTemplate();

// ... configure transactionTemplate ...

SessionRepository<? extends Session> repository = new
JdbcIndexedSessionRepository(jdbcTemplate,
        transactionTemplate);
```

For additional information on how to create and configure `JdbcTemplate` and `PlatformTransactionManager`, see the Spring Framework Reference Documentation.

9.11.2. Using `@EnableJdbcHttpSession`

In a web environment, the simplest way to create a new `JdbcIndexedSessionRepository` is to use `@EnableJdbcHttpSession`. You can find complete example usage in the Samples and Guides (Start Here) You can use the following attributes to customize the configuration:

- **tableName**: The name of database table used by Spring Session to store sessions
- **maxInactiveIntervalInSeconds**: The amount of time before the session will expire in seconds

Customizing `LobHandler`

You can customize BLOB handling by creating a bean named `springSessionLobHandler` that implements `LobHandler`.

Customizing `ConversionService`

You can customize the default serialization and deserialization of the session by providing a `ConversionService` instance. When working in a typical Spring environment, the default `ConversionService` bean (named `conversionService`) is automatically picked up and used for serialization and deserialization. However, you can override the default `ConversionService` by providing a bean named `springSessionConversionService`.

9.11.3. Storage Details

By default, this implementation uses `SPRING_SESSION` and `SPRING_SESSION_ATTRIBUTES` tables to store sessions. Note that you can customize the table name, as already described. In that case, the table used to store attributes is named by using the provided table name suffixed with `_ATTRIBUTES`. If further customizations are needed, you can customize the SQL queries used by the repository by using `set*Query` setter methods. In this case, you need to manually configure the `sessionRepository` bean.

Due to the differences between the various database vendors, especially when it comes to storing

binary data, make sure to use SQL scripts specific to your database. Scripts for most major database vendors are packaged as `org/springframework/session/jdbc/schema-*.sql`, where `*` is the target database type.

For example, with PostgreSQL, you can use the following schema script:

```
CREATE TABLE SPRING_SESSION (
    PRIMARY_ID CHAR(36) NOT NULL,
    SESSION_ID CHAR(36) NOT NULL,
    CREATION_TIME BIGINT NOT NULL,
    LAST_ACCESS_TIME BIGINT NOT NULL,
    MAX_INACTIVE_INTERVAL INT NOT NULL,
    EXPIRY_TIME BIGINT NOT NULL,
    PRINCIPAL_NAME VARCHAR(100),
    CONSTRAINT SPRING_SESSION_PK PRIMARY KEY (PRIMARY_ID)
);

CREATE UNIQUE INDEX SPRING_SESSION_IX1 ON SPRING_SESSION (SESSION_ID);
CREATE INDEX SPRING_SESSION_IX2 ON SPRING_SESSION (EXPIRY_TIME);
CREATE INDEX SPRING_SESSION_IX3 ON SPRING_SESSION (PRINCIPAL_NAME);

CREATE TABLE SPRING_SESSION_ATTRIBUTES (
    SESSION_PRIMARY_ID CHAR(36) NOT NULL,
    ATTRIBUTE_NAME VARCHAR(200) NOT NULL,
    ATTRIBUTE_BYTES BYTEA NOT NULL,
    CONSTRAINT SPRING_SESSION_ATTRIBUTES_PK PRIMARY KEY (SESSION_PRIMARY_ID,
ATTRIBUTE_NAME),
    CONSTRAINT SPRING_SESSION_ATTRIBUTES_FK FOREIGN KEY (SESSION_PRIMARY_ID)
REFERENCES SPRING_SESSION(PRIMARY_ID) ON DELETE CASCADE
);
```

With MySQL database, you can use the following script:

```
CREATE TABLE SPRING_SESSION (
    PRIMARY_ID CHAR(36) NOT NULL,
    SESSION_ID CHAR(36) NOT NULL,
    CREATION_TIME BIGINT NOT NULL,
    LAST_ACCESS_TIME BIGINT NOT NULL,
    MAX_INACTIVE_INTERVAL INT NOT NULL,
    EXPIRY_TIME BIGINT NOT NULL,
    PRINCIPAL_NAME VARCHAR(100),
    CONSTRAINT SPRING_SESSION_PK PRIMARY KEY (PRIMARY_ID)
) ENGINE=InnoDB ROW_FORMAT=DYNAMIC;

CREATE UNIQUE INDEX SPRING_SESSION_IX1 ON SPRING_SESSION (SESSION_ID);
CREATE INDEX SPRING_SESSION_IX2 ON SPRING_SESSION (EXPIRY_TIME);
CREATE INDEX SPRING_SESSION_IX3 ON SPRING_SESSION (PRINCIPAL_NAME);

CREATE TABLE SPRING_SESSION_ATTRIBUTES (
    SESSION_PRIMARY_ID CHAR(36) NOT NULL,
    ATTRIBUTE_NAME VARCHAR(200) NOT NULL,
    ATTRIBUTE_BYTES BLOB NOT NULL,
    CONSTRAINT SPRING_SESSION_ATTRIBUTES_PK PRIMARY KEY (SESSION_PRIMARY_ID,
ATTRIBUTE_NAME),
    CONSTRAINT SPRING_SESSION_ATTRIBUTES_FK FOREIGN KEY (SESSION_PRIMARY_ID)
REFERENCES SPRING_SESSION(PRIMARY_ID) ON DELETE CASCADE
) ENGINE=InnoDB ROW_FORMAT=DYNAMIC;
```

9.11.4. Transaction Management

All JDBC operations in `JdbcIndexedSessionRepository` are performed in a transactional manner. Transactions are performed with propagation set to `REQUIRES_NEW` in order to avoid unexpected behavior due to interference with existing transactions (for example, running a `save` operation in a thread that already participates in a read-only transaction).

9.12. Using `HazelcastIndexedSessionRepository`

`HazelcastIndexedSessionRepository` is a `SessionRepository` implementation that stores sessions in Hazelcast's distributed `IMap`. In a web environment, this is typically used in combination with `SessionRepositoryFilter`.

9.12.1. Instantiating a `HazelcastIndexedSessionRepository`

The following example shows how to create a new instance:

```
Config config = new Config();

// ... configure Hazelcast ...

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance(config);

HazelcastIndexedSessionRepository repository = new
HazelcastIndexedSessionRepository(hazelcastInstance);
```

For additional information on how to create and configure Hazelcast instance, see the Hazelcast documentation.

9.12.2. Using `@EnableHazelcastHttpSession`

To use Hazelcast as your backing source for the `SessionRepository`, you can add the `@EnableHazelcastHttpSession` annotation to a `@Configuration` class. Doing so extends the functionality provided by the `@EnableSpringHttpSession` annotation but makes the `SessionRepository` for you in Hazelcast. You must provide a single `HazelcastInstance` bean for the configuration to work. You can find a complete configuration example in the Samples and Guides (Start Here).

9.12.3. Basic Customization

You can use the following attributes on `@EnableHazelcastHttpSession` to customize the configuration:

- **maxInactiveIntervalInSeconds**: The amount of time before the session expires, in seconds. The default is 1800 seconds (30 minutes)
- **sessionMapName**: The name of the distributed `Map` that is used in Hazelcast to store the session data.

9.12.4. Session Events

Using a `MapListener` to respond to entries being added, evicted, and removed from the distributed `Map` causes these events to trigger publishing of `SessionCreatedEvent`, `SessionExpiredEvent`, and `SessionDeletedEvent` events (respectively) through the `ApplicationEventPublisher`.

9.12.5. Storage Details

Sessions are stored in a distributed `IMap` in Hazelcast. The `IMap` interface methods are used to `get()` and `put()` Sessions. Additionally, the `values()` method supports a `FindByIndexNameSessionRepository#findByIndexNameAndIndexValue` operation, together with appropriate `ValueExtractor` (which needs to be registered with Hazelcast). See the Hazelcast Spring Sample for more details on this configuration. The expiration of a session in the `IMap` is handled by Hazelcast's support for setting the time to live on an entry when it is `put()` into the `IMap`. Entries (sessions) that have been idle longer than the time to live are automatically removed from the `IMap`.

You should not need to configure any settings such as `max-idle-seconds` or `time-to-live-seconds` for

the `IMap` within the Hazelcast configuration.

Note that if you use Hazelcast's `MapStore` to persist your sessions `IMap`, the following limitations apply when reloading the sessions from `MapStore`:

- Reloading triggers `EntryAddedListener` results in `SessionCreatedEvent` being re-published
- Reloading uses default TTL for a given `IMap` results in sessions losing their original TTL

9.13. Using `CookieSerializer`

A `CookieSerializer` is responsible for defining how the session cookie is written. Spring Session comes with a default implementation using `DefaultCookieSerializer`.

9.13.1. Exposing `CookieSerializer` as a bean

Exposing the `CookieSerializer` as a Spring bean augments the existing configuration when you use configurations like `@EnableRedisHttpSession`.

The following example shows how to do so:

```
@Bean
public CookieSerializer cookieSerializer() {
    DefaultCookieSerializer serializer = new DefaultCookieSerializer();
    serializer.setCookieName("JSESSIONID"); ①
    serializer.setCookiePath("/"); ②
    serializer.setDomainNamePattern("^.+?\\.(\\w+\\.[a-z]+)$"); ③
    return serializer;
}
```

① We customize the name of the cookie to be `JSESSIONID`.

② We customize the path of the cookie to be `/` (rather than the default of the context root).

③ We customize the domain name pattern (a regular expression) to be `^.+?\\.(\\w+\\.[a-z]+)$`. This allows sharing a session across domains and applications. If the regular expression does not match, no domain is set and the existing domain is used. If the regular expression matches, the first grouping is used as the domain. This means that a request to https://child.example.com sets the domain to `example.com`. However, a request to http://localhost:8080/ or https://192.168.1.100:8080/ leaves the cookie unset and, thus, still works in development without any changes being necessary for production.

You should only match on valid domain characters, since the domain name is reflected in the response. Doing so prevents a malicious user from performing such attacks as HTTP Response Splitting.

9.13.2. Customizing `CookieSerializer`

You can customize how the session cookie is written by using any of the following configuration options on the `DefaultCookieSerializer`.

- `cookieName`: The name of the cookie to use. Default: `SESSION`.
- `useSecureCookie`: Specifies whether a secure cookie should be used. Default: Use the value of `HttpServletRequest.isSecure()` at the time of creation.
- `cookiePath`: The path of the cookie. Default: The context root.
- `cookieMaxAge`: Specifies the max age of the cookie to be set at the time the session is created. Default: `-1`, which indicates the cookie should be removed when the browser is closed.
- `jvmRoute`: Specifies a suffix to be appended to the session ID and included in the cookie. Used to identify which JVM to route to for session affinity. With some implementations (that is, Redis) this option provides no performance benefit. However, it can help with tracing logs of a particular user.
- `domainName`: Allows specifying a specific domain name to be used for the cookie. This option is simple to understand but often requires a different configuration between development and production environments. See `domainNamePattern` as an alternative.
- `domainNamePattern`: A case-insensitive pattern used to extract the domain name from the `HttpServletRequest#getServerName()`. The pattern should provide a single grouping that is used to extract the value of the cookie domain. If the regular expression does not match, no domain is set and the existing domain is used. If the regular expression matches, the first grouping is used as the domain.
- `sameSite`: The value for the `SameSite` cookie directive. To disable the serialization of the `SameSite` cookie directive, you may set this value to `null`. Default: `Lax`

You should only match on valid domain characters, since the domain name is reflected in the response. Doing so prevents a malicious user from performing such attacks as HTTP Response Splitting.

Chapter 10. Customizing `SessionRepository`

Implementing a custom `SessionRepository` API should be a fairly straightforward task. Coupling the custom implementation with `@EnableSpringHttpSession` support lets you reuse existing Spring Session configuration facilities and infrastructure. There are, however, a couple of aspects that deserve closer consideration.

During the lifecycle of an HTTP request, the `HttpSession` is typically persisted to `SessionRepository` twice. The first persist operation is to ensure that the session is available to the client as soon as the client has access to the session ID, and it is also necessary to write after the session is committed because further modifications to the session might be made. Having this in mind, we generally recommend that a `SessionRepository` implementation keep track of changes to ensure that only deltas are saved. This is particularly important in highly concurrent environments, where multiple requests operate on the same `HttpSession` and, therefore, cause race conditions, with requests overriding each other's changes to session attributes. All of the `SessionRepository` implementations provided by Spring Session use the described approach to persist session changes and can be used for guidance when you implement custom `SessionRepository`.

Note that the same recommendations apply for implementing a custom `ReactiveSessionRepository` as well. In this case, you should use the `@EnableSpringWebSession`.

Chapter 11. Upgrading to 2.x

With the new major release version, the Spring Session team took the opportunity to make some non-passive changes. The focus of these changes is to improve and harmonize Spring Session's APIs as well as remove the deprecated components.

11.1. Baseline Update

Spring Session 2.0 requires Java 8 and Spring Framework 5.0 as a baseline, since its entire codebase is now based on Java 8 source code. See Upgrading to Spring Framework 5.x for more on upgrading Spring Framework.

11.2. Replaced and Removed Modules

As a part of the project's splitting of the modules, the existing `spring-session` has been replaced with the `spring-session-core` module. The `spring-session-core` module holds only the common set of APIs and components, while other modules contain the implementation of the appropriate `SessionRepository` and functionality related to that data store. This applies to several existing modules that were previously a simple dependency aggregator helper module. With new module arrangement, the following modules actually carry the implementation:

- Spring Session Data Redis
- Spring Session JDBC
- Spring Session Hazelcast

Also, the following modules were removed from the main project repository:

- Spring Session Data MongoDB
- Spring Session Data GemFire

Note that these two have moved to separate repositories and continue to be available under new artifact names:

- `spring-session-data-mongodb`
- `spring-session-data-geode`

11.3. Replaced and Removed Packages, Classes, and Methods

The following changes were made to packages, classes, and methods:

- `ExpiringSession` API has been merged into the `Session` API.
- The `Session` API has been enhanced to make full use of Java 8.
- The `Session` API has been extended with `changeSessionId` support.

- The `SessionRepository` API has been updated to better align with Spring Data method naming conventions.
- `AbstractSessionEvent` and its subclasses are no longer constructable without an underlying `Session` object.
- The Redis namespace used by `RedisOperationsSessionRepository` is now fully configurable, instead of being partially configurable.
- Redis configuration support has been updated to avoid registering a Spring Session-specific `RedisTemplate` bean.
- JDBC configuration support has been updated to avoid registering a Spring Session-specific `JdbcTemplate` bean.
- Previously deprecated classes and methods have been removed across the codebase

11.4. Dropped Support

As a part of the changes to `HttpSessionStrategy` and its alignment to the counterpart from the reactive world, the support for managing multiple users' sessions in a single browser instance has been removed. The introduction of a new API to replace this functionality is under consideration for future releases.

Chapter 12. Spring Session Community

We are glad to consider you a part of our community. The following sections provide additional about how to interact with the Spring Session community.

12.1. Support

You can get help by asking questions on Stack Overflow with the `spring-session` tag. Similarly, we encourage helping others by answering questions on Stack Overflow.

12.2. Source Code

You can find the source code on GitHub at https://github.com/spring-projects/spring-session/

12.3. Issue Tracking

We track issues in GitHub issues at https://github.com/spring-projects/spring-session/issues

12.4. Contributing

We appreciate pull requests.

12.5. License

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

12.6. Community Extensions

Name	Location
Spring Session Infinispan	https://infinispan.org/infinispan-spring-boot/master/spring_boot_starter.html#_enabling_spring_session_support

Chapter 13. Minimum Requirements

The minimum requirements for Spring Session are:

- Java 8+.
- If you run in a Servlet Container (not required), Servlet 3.1+.
- If you use other Spring libraries (not required), the minimum required version is Spring 5.0.x.
- `@EnableRedisHttpSession` requires Redis 2.8+. This is necessary to support Session Expiration
- `@EnableHazelcastHttpSession` requires Hazelcast 3.6+. This is necessary to support `FindByIndexNameSessionRepository`

At its core, Spring Session has a required dependency only on `spring-jcl`. For an example of using Spring Session without any other Spring dependencies, see the hazelcast sample application.