OAuth2 Autoconfig

Table of Contents

If you have `spring-security-oauth2` on your classpath you can take advantage of some auto-configuration to make it easy to set up Authorization or Resource Server. For full details, see the Spring Security OAuth 2 Developers Guide.

> **Note**
>
> This project is a port of the Spring Security OAuth support that came with Spring Boot 1.x. Support was removed in favor of Spring Security 5's first class OAuth support. To ease migration, this project exists as a bridge between the old Spring Security OAuth support and Spring Boot 2.x.

1. Downloading

Since `spring-security-oauth2-autoconfigure` is externalized you will need to ensure to add it to your classpath.

1.1 Source

You can get the source and log issues on GitHub.

1.2 Maven

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

pom.xml.

```
<dependencies>
  <!-- ... other dependency elements ... -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.security.oauth.boot</groupId>
    <artifactId>spring-security-oauth2-autoconfigure</artifactId>
    <version>2.1.15.RELEASE</version>
  </dependency>
</dependencies>
```

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

If you are using a SNAPSHOT version, you will need to ensure you have the Spring Snapshot repository defined as shown below:

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 are using a milestone or release candidate version, you will need to ensure you have the Spring Milestone repository defined as shown below:

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>
```

1.3 Gradle

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

build.gradle.

```
dependencies {
  compile 'org.springframework.boot:spring-boot-starter-security'
  compile 'org.springframework.security.oauth.boot:spring-security-oauth2-autoconfigure:2.1.15.RELEASE'
}
```

All GA releases (i.e. versions ending in .RELEASE) are deployed to Maven Central, so using the mavenCentral() repository is sufficient for GA releases.

build.gradle.

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

If you are using a SNAPSHOT version, you will need to ensure you have the Spring Snapshot repository defined as shown below:

build.gradle.

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

If you are using a milestone or release candidate version, you will need to ensure you have the Spring Milestone repository defined as shown below:

build.gradle.

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

2. Authorization Server

To create an Authorization Server and grant access tokens you need to use `@EnableAuthorizationServer` and provide `security.oauth2.client.client-id` and `security.oauth2.client.client-secret]` properties. The client will be registered for you in an in-memory repository.

Having done that you will be able to use the client credentials to create an access token, for example:

```
$ curl client:secret@localhost:8080/oauth/token -d grant_type=password -d username=user -d password=pwd
```

The basic auth credentials for the `/token` endpoint are the `client-id` and `client-secret`. The user credentials are the normal Spring Security user details (which default in Spring Boot to “user” and a random password).

To switch off the auto-configuration and configure the Authorization Server features yourself just add a `@Bean` of type `AuthorizationServerConfigurer`.

If you use your own authorization server configuration to configure the list of valid clients through an instance of `ClientDetailsServiceConfigurer` as shown below, take note that the passwords you configure here are subject to [the modernized password storage](#) that came with Spring Security 5. That means you have to prefix your passwords with an Id if you use Spring Boot Securities defaults for password storage.

```
@Component
public class CustomAuthorizationServerConfigurer extends
    AuthorizationServerConfigurerAdapter {

    AuthenticationManager authenticationManager;

    public CustomAuthorizationServerConfigurer(
        AuthenticationConfiguration authenticationConfiguration
    ) throws Exception {
        this.authenticationManager =
            authenticationConfiguration.getAuthenticationManager();
    }

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients
    ) throws Exception {
        clients.inMemory()
            .withClient("client")
                .authorizedGrantTypes("password")
                .secret("{noop}secret")
                .scopes("all");
    }

    @Override
    public void configure(
        AuthorizationServerEndpointsConfigurer endpoints
    ) throws Exception {
        endpoints.authenticationManager(authenticationManager);
    }
}
```

3. Resource Server

To use the access token you need a Resource Server (which can be the same as the Authorization Server). Creating a Resource Server is easy, just add `@EnableResourceServer` and provide some configuration to allow the server to decode access tokens. If your application is also an Authorization Server it already knows how to decode tokens, so there is nothing else to do. If your app is a standalone service then you need to give it some more configuration, one of the following options:

- `security.oauth2.resource.user-info-uri` to use the `/me` resource (e.g. `https://uaa.run.pivotal.io/userinfo` on Pivotal Web Services (PWS))
- `security.oauth2.resource.token-info-uri` to use the token decoding endpoint (e.g. `https://uaa.run.pivotal.io/check_token` on PWS).

If you specify both the `user-info-uri` and the `token-info-uri` then you can set a flag to say that one is preferred over the other (`prefer-token-info=true` is the default).

Alternatively (instead of `user-info-uri` or `token-info-uri`) if the tokens are JWTs you can configure a `security.oauth2.resource.jwt.key-value` to decode them locally (where the key is a verification key). The verification key value is either a symmetric secret or PEM-encoded RSA public key. If you don't have the key and it's public you can provide a URI where it can be downloaded (as a JSON object with a "value" field) with `security.oauth2.resource.jwt.key-uri`. E.g. on PWS:

```
$ curl https://uaa.run.pivotal.io/token_key
{"alg":"SHA256withRSA","value":"-----BEGIN PUBLIC KEY-----\nMIIBI...\n-----END PUBLIC KEY-----\n"}
```

Additionally, if your authorization server has an endpoint that returns a set of JSON Web Keys(JWKs), you can configure `security.oauth2.resource.jwk.key-set-uri`. E.g. on PWS:

```
$ curl https://uaa.run.pivotal.io/token_keys
{"keys":[{"kid":"key-1","alg":"RS256","value":"-----BEGIN PUBLIC KEY-----\nMIIBI...\n-----END PUBLIC
 KEY-----\n"]}
```

> **Note**
>
> Configuring both JWT and JWK properties will cause an error. Only one of `security.oauth2.resource.jwt.key-uri` (or `security.oauth2.resource.jwt.key-value`) and `security.oauth2.resource.jwk.key-set-uri` should be configured.

> **Warning**
>
> If you use the `security.oauth2.resource.jwt.key-uri` or `security.oauth2.resource.jwk.key-set-uri`, the authorization server needs to be running when your application starts up. It will log a warning if it can't find the key, and tell you what to do to fix it.

OAuth2 resources are protected by a filter chain with order `security.oauth2.resource.filter-order`.

By default the filters in `AuthorizationServerConfigurerAdapter` come first, followed by those in `ResourceServerConfigurerAdapter`, followed by those in `WebSecurityConfigurerAdapter`.

This means that **all application endpoints will require bearer token authentication** unless one of two things happens:

1. The filter chain order is changed or
2. The `ResourceServerConfigurerAdapter` set of authorized requests is narrowed

The first, changing the filter chain order, can be done by moving `WebSecurityConfigurerAdapter` in front of `ResourceServerConfigurerAdapter` like so:

```
@Order(2)
@EnableWebSecurity
public WebSecurityConfig extends WebSecurityConfigurerAdapter {
        // ...
}
```

> **Note**
>
> Resource Server's default `@Order` value is 3 which is why the example sets Web's `@Order` to 2, so that it's evaluated earlier.

While this may work, it's a little odd since we may simply trade one problem:

> `ResourceServerConfigurerAdapter` is handling requests it shouldn't

For another:

> `WebSecurityConfigurerAdapter` is handling requests it shouldn't

The more robust solution, then, is to indicate to `ResourceServerConfigurerAdapter` which endpoints should be secured by bearer token authentication.

For example, the following configures Resource Server to secure the web application endpoints that begin with `/rest`:

```
@EnableResourceServer
public ResourceServerConfig extends ResourceServerConfigurerAdapter {
        @Override
    protected void configure(HttpSecurity http) {
        http
            .requestMatchers()
                .antMatchers("/rest/**")
            .authorizeRequests()
                .anyRequest().authenticated();
    }
}
```

Part I. Token Type in User Info

Google, and certain other 3rd party identity providers, are more strict about the token type name that is sent in the headers to the user info endpoint. The default is “Bearer” which suits most providers and matches the spec, but if you need to change it you can set `security.oauth2.resource.token-type`.

Part II. Customizing the User Info RestTemplate

If you have a `user-info-uri`, the resource server features use an `OAuth2RestTemplate` internally to fetch user details for authentication. This is provided as a `@Bean` of type `UserInfoRestTemplateFactory`. The default should be fine for most providers, but occasionally you might need to add additional interceptors, or change the request authenticator (which is how the token gets attached to outgoing requests). To add a customization just create a bean of type `UserInfoRestTemplateCustomizer` - it has a single method that will be called after the bean is created but before it is initialized. The rest template that is being customized here is *only* used internally to carry out authentication. Alternatively, you could define your own `UserInfoRestTemplateFactory` `@Bean` to take full control.

> **Tip**
>
> To set an RSA key value in YAML use the “pipe” continuation marker to split it over multiple lines (“|”) and remember to indent the key value (it’s a standard YAML language feature). Example:

```
security:
  oauth2:
    resource:
      jwt:
        keyValue: |
          -----BEGIN PUBLIC KEY-----
          MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC...
          -----END PUBLIC KEY-----
```

4. Client

To make your web-app into an OAuth2 client you can simply add `@EnableOAuth2Client` and Spring Boot will create an `OAuth2ClientContext` and `OAuth2ProtectedResourceDetails` that are necessary to create an `OAuth2RestOperations`. Spring Boot does not automatically create such bean but you can easily create your own:

```
@Bean
public OAuth2RestTemplate oauth2RestTemplate(OAuth2ClientContext oauth2ClientContext,
        OAuth2ProtectedResourceDetails details) {
    return new OAuth2RestTemplate(details, oauth2ClientContext);
}
```

> **Note**
>
> You may want to add a qualifier and review your configuration as more than one `RestTemplate` may be defined in your application.

This configuration uses `security.oauth2.client.*` as credentials (the same as you might be using in the Authorization Server), but in addition it will need to know the authorization and token URIs in the Authorization Server. For example:

application.yml.

```
security:
  oauth2:
    client:
      clientId: bd1c0a783ccdd1c9b9e4
      clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1
      accessTokenUri: https://github.com/login/oauth/access_token
      userAuthorizationUri: https://github.com/login/oauth/authorize
      clientAuthenticationScheme: form
```

An application with this configuration will redirect to Github for authorization when you attempt to use the `OAuth2RestTemplate`. If you are already signed into Github you won't even notice that it has authenticated. These specific credentials will only work if your application is running on port 8080 (register your own client app in Github or other provider for more flexibility).

To limit the scope that the client asks for when it obtains an access token you can set `security.oauth2.client.scope` (comma separated or an array in YAML). By default the scope is empty and it is up to Authorization Server to decide what the defaults should be, usually depending on the settings in the client registration that it holds.

> **Note**
>
> There is also a setting for `security.oauth2.client.client-authentication-scheme` which defaults to "header" (but you might need to set it to "form" if, like Github for instance, your OAuth2 provider doesn't like header authentication). In fact, the `security.oauth2.client.*` properties are bound to an instance of `AuthorizationCodeResourceDetails` so all its properties can be specified.

> **Tip**
>
> In a non-web application you can still create an `OAuth2RestOperations` and it is still wired into the `security.oauth2.client.*` configuration. In this case it is a "client credentials token

> grant” you will be asking for if you use it (and there is no need to use `@EnableOAuth2Client` or `@EnableOAuth2Sso`). To prevent that infrastructure to be defined, just remove the `security.oauth2.client.client-id` from your configuration (or make it the empty string).

5. Single Sign On

An OAuth2 Client can be used to fetch user details from the provider (if such features are available) and then convert them into an `Authentication` token for Spring Security. The Resource Server above support this via the `user-info-uri` property This is the basis for a Single Sign On (SSO) protocol based on OAuth2, and Spring Boot makes it easy to participate by providing an annotation `@EnableOAuth2Sso`. The Github client above can protect all its resources and authenticate using the Github `/user/` endpoint, by adding that annotation and declaring where to find the endpoint (in addition to the `security.oauth2.client.*` configuration already listed above):

application.yml.

```
security:
  oauth2:
# ...
  resource:
    userInfoUri: https://api.github.com/user
    preferTokenInfo: false
```

Since all paths are secure by default, there is no “home” page that you can show to unauthenticated users and invite them to login (by visiting the `/login` path, or the path specified by `security.oauth2.sso.login-path`).

To customize the access rules or paths to protect, so you can add a “home” page for instance, `@EnableOAuth2Sso` can be added to a `WebSecurityConfigurerAdapter` and the annotation will cause it to be decorated and enhanced with the necessary pieces to get the `/login` path working. For example, here we simply allow unauthenticated access to the home page at "/" and keep the default for everything else:

```
@Configuration
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .mvcMatchers("/").permitAll()
                .anyRequest().authenticated();
    }
}
```

Also note that since all endpoints are secure by default, this includes any default error handling endpoints, for example, the endpoint "/error". This means that if there is some problem during Single Sign On that requires the application to redirect to the "/error" page, then this can cause an infinite redirect between the identity provider and the receiving application.

First, think carefully about making an endpoint insecure as you may find that the behavior is simply evidence of a different problem. However, this behavior can be addressed by configuring the application to permit "/error":

```
@Configuration
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/error").permitAll()
                .anyRequest().authenticated();
    }
}
```

Appendix A. Common application properties

Various properties can be specified inside your `application.properties`/`application.yml` file or as command line switches. This section provides a list of common Spring Boot properties and references to the underlying classes that consume them.

> **Note**
>
> Property contributions can come from additional jar files on your classpath so you should not consider this an exhaustive list. It is also perfectly legit to define your own properties.

> **Warning**
>
> This sample file is meant as a guide only. Do **not** copy/paste the entire content into your application; rather pick only the properties that you need.

```
# SECURITY OAUTH2 CLIENT (OAuth2ClientProperties)
security.oauth2.client.client-id= # OAuth2 client id.
security.oauth2.client.client-secret= # OAuth2 client secret. A random secret is generated by default

# SECURITY OAUTH2 RESOURCES (ResourceServerProperties)
security.oauth2.resource.id= # Identifier of the resource.
security.oauth2.resource.jwt.key-uri= # The URI of the JWT token. Can be set if the value is not
 available and the key is public.
security.oauth2.resource.jwt.key-value= # The verification key of the JWT token. Can either be a
 symmetric secret or PEM-encoded RSA public key.
security.oauth2.resource.jwk.key-set-uri= # The URI for getting the set of keys that can be used to
 validate the token.
security.oauth2.resource.prefer-token-info=true # Use the token info, can be set to false to use the
 user info.
security.oauth2.resource.service-id=resource #
security.oauth2.resource.token-info-uri= # URI of the token decoding endpoint.
security.oauth2.resource.token-type= # The token type to send when using the userInfoUri.
security.oauth2.resource.user-info-uri= # URI of the user endpoint.

# SECURITY OAUTH2 SSO (OAuth2SsoProperties)
security.oauth2.sso.login-path=/login # Path to the login page, i.e. the one that triggers the redirect
 to the OAuth2 Authorization Server
```