OAuth2 Boot

Table of Contents

If you have `spring-security-oauth2` on your classpath, you can take advantage of some auto-configuration to simplify setting up Authorization and Resource Servers. For full details, see the Spring Security OAuth 2 Developers Guide.

> **Note**
>
> **The following projects are in maintenance mode:**
>
> - `spring-security-oauth2`
> - `spring-security-oauth2-autoconfigure`
>
> You are, of course, welcome to use them, and we will help you out!
>
> **However, before selecting `spring-security-oauth2` and `spring-security-oauth2-autoconfigure`, you should check out Spring Security's feature matrix to see if the new first-class support meets your needs.**

> **Note**
>
> This project is a port of the Spring Security OAuth support that came with Spring Boot 1.x. **Support was removed in Spring Boot 2.x 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. Authorization Server

Spring Security OAuth2 Boot simplifies standing up an OAuth 2.0 Authorization Server.

1.1 Do I Need to Stand Up My Own Authorization Server?

You need to stand up your own authorization server if:

- You want to delegate the operations of sign-in, sign-out, and password recovery to a separate service (also called *identity federation*) that you want to manage yourself and
- You want to use the OAuth 2.0 protocol for this separate service to coordinate with other services

1.2 Dependencies

To use the auto-configuration features in this library, you need `spring-security-oauth2`, which has the OAuth 2.0 primitives and `spring-security-oauth2-autoconfigure`. Note that you need to specify the version for `spring-security-oauth2-autoconfigure`, since it is not managed by Spring Boot any longer, though it should match Boot's version anyway.

For JWT support, you also need `spring-security-jwt`.

1.3 Minimal OAuth2 Boot Configuration

Creating a minimal Spring Boot authorization server consists of three basic steps:

1. Including the dependencies.
2. Including the `@EnableAuthorizationServer` annotation.
3. Specifying at least one client ID and secret pair.

Enabling the Authorization Server

Similar to other Spring Boot `@Enable` annotations, you can add the `@EnableAuthorizationServer` annotation to the class that contains your `main` method, as the following example shows:

```
@EnableAuthorizationServer
@SpringBootApplication
public class SimpleAuthorizationServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(SimpleAuthorizationServerApplication, args);
    }
}
```

Adding this annotation imports other Spring configuration files that add a number of reasonable defaults, such as how tokens ought to be signed, their duration, and what grants to allow.

Specifying a Client and Secret

By spec, numerous OAuth 2.0 endpoints require client authentication, so you need to specify at least one client in order for anyone to be able to communicate with your authorization server.

The following example shows how to specify a client:

```
security:
  oauth2:
    client:
      client-id: first-client
      client-secret: noonewilleverguess
```

> **Note**
>
> While convenient, this makes a number of assumptions that are unlikely to be viable in production. You likely need to do more than this to ship.

That's it! But, what do you do with it? We cover that next.

Retrieving a Token

OAuth 2.0 is essentially a framework that specifies strategies for exchanging long-lived tokens for short-lived ones.

By default, `@EnableAuthorizationServer` grants a client access to client credentials, which means you can do something like the following:

```
curl first-client:noonewilleverguess@localhost:8080/oauth/token -dgrant_type=client_credentials -
dscope=any
```

The application responds with a token similar to the following:

```
{
    "access_token" : "f05a1ea7-4c80-4583-a123-dc7a99415588",
    "token_type" : "bearer",
    "expires_in" : 43173,
    "scope" : "any"
}
```

This token can be presented to any resource server that supports opaque OAuth 2.0 tokens and is configured to point at this authorization server for verification.

From here, you can jump to:

- Section 1.4, "How to Switch Off OAuth2 Boot's Auto Configuration"
- Section 1.5, "How to Make Authorization Code Grant Flow Work"
- Section 1.6, "How to Make Password Grant Flow Work"
- Section 1.7, "How and When to Give Authorization Server an AuthenticationManager"
- Section 1.8, "Is Authorization Server Compatible with Spring Security 5.1 Resource Server and Client?"
- How to Configure for Jwt Tokens

1.4 How to Switch Off OAuth2 Boot's Auto Configuration

Basically, the OAuth2 Boot project creates an instance of `AuthorizationServerConfigurer` with some reasonable defaults:

- It registers a `NoOpPasswordEncoder` (overriding the Spring Security default)
- It lets the client you provided use any grant type this server supports: `authorization_code`, `password`, `client_credentials`, `implicit`, or `refresh_token`.

Otherwise, it also tries to pick up a handful of beans, if they are defined — namely:

- `AuthenticationManager`: For looking up end users (not clients)
- `TokenStore`: For generating and retrieving tokens
- `AccessTokenConverter`: For converting access tokens into different formats, such as JWT.

> **Note**
>
> While this documentation covers a bit of what each of these beans does, the Spring Security OAuth documentation is a better place to read up on its primitives

If you expose a bean of type `AuthorizationServerConfigurer`, none of this is done automatically.

So, for example, if you need to configure more than one client, change their allowed grant types, or use something better than the no-op password encoder (highly recommended!), then you want to expose your own `AuthorizationServerConfigurer`, as the following example shows:

```
@Configuration
public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

    @Autowired DataSource dataSource;

    protected void configure(ClientDetailsServiceConfigurer clients) {
        clients
            .jdbc(this.dataSource)
            .passwordEncoder(PasswordEncoderFactories.createDelegatingPasswordEncoder());
    }
}
```

The preceding configuration causes OAuth2 Boot to no longer retrieve the client from environment properties and now falls back to the Spring Security password encoder default.

From here, you may want to learn more about:

- Section 1.5, “How to Make Authorization Code Grant Flow Work”
- Section 1.6, “How to Make Password Grant Flow Work”

1.5 How to Make Authorization Code Grant Flow Work

With the default configuration, while the Authorization Code Flow is technically allowed, it is not completely configured.

This is because, in addition to what comes pre-configured, the Authorization Code Flow requires:

- End users
- An end-user login flow, and
- A redirect URI registered with the client

Adding End Users

In a typical Spring Boot application secured by Spring Security, users are defined by a `UserDetailsService`. In that regard, an authorization server is no different, as the following example shows:

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    @Override
    public UserDetailsService userDetailsService() {
        return new InMemoryUserDetailsManager(
            User.withDefaultPasswordEncoder()
                .username("enduser")
                .password("password")
                .roles("USER")
                .build());
    }
}
```

Note that, as is typical of a Spring Security web application, users are defined in a `WebSecurityConfigurerAdapter` instance.

Adding an End-User Login Flow

Incidentally, adding an instance of `WebSecurityConfigurerAdapter` is all we need for now to add a form login flow for end users. However, note that this is where any other configuration regarding the web application itself, not the OAuth 2.0 API, goes.

If you want to customize the login page, offer more than just form login for the user, or add additional support like password recovery, the `WebSecurityConfigurerAdapter` picks it up.

Registering a Redirect URI With the Client

OAuth2 Boot does not support configuring a redirect URI as a property — say, alongside `client-id` and `client-secret`.

To add a redirect URI, you need to specify the client by using either `InMemoryClientDetailsService` or `JdbcClientDetailsService`.

Doing either means replacing the OAuth2 Boot-provided `AuthorizationServerConfigurer` with your own, as the following example shows:

```
@Configuration
public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

    @Bean
    PasswordEncoder passwordEncoder() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

    protected void configure(ClientDetailsServiceConfigurer clients) {
        clients
            .inMemory()
                .withClient("first-client")
                .secret(passwordEncoder().encode("noonewilleverguess"))
                .scopes("resource:read")
                .authorizedGrantTypes("authorization_code")
                .redirectUris("http://localhost:8081/oauth/login/client-app");
    }
}
```

Testing Authorization Code Flow

Testing OAuth can be tricky since it requires more than one server to see the full flow in action. However, the first steps are straight-forward:

1. Browse to http://localhost:8080/oauth/authorize?grant_type=authorization_code&response_type=code&client_id=first-client&state=1234
2. The application, if the user is not logged in, redirects to the login page, at http://localhost:8080/login
3. Once the user logs in, the application generates a code and redirects to the registered redirect URI — in this case, http://localhost:8081/oauth/login/client-app

The flow could continue at this point by standing up any resource server that is configured for opaque tokens and is pointed at this authorization server instance.

1.6 How to Make Password Grant Flow Work

With the default configuration, while the Password Flow is technically possible, it, like Authorization Code, is missing users.

That said, because the default configuration creates a user with a username of `user` and a randomly-generated password, you can hypothetically check the logs for the password and do the following:

```
curl first-client:noonewilleverguess@localhost:8080/oauth/token -dgrant_type=password -dscope=any -
dusername=user -dpassword=the-password-from-the-logs
```

When you run that command, you should get a token back.

More likely, though, you want to specify a set of users.

As was stated in Section 1.5, “How to Make Authorization Code Grant Flow Work”, in Spring Security, users are typically specified in a `UserDetailsService` and this application is no different, as the following example shows:

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    @Override
    public UserDetailsService userDetailsService() {
        return new InMemoryUserDetailsManager(
            User.withDefaultPasswordEncoder()
                .username("enduser")
                .password("password")
                .roles("USER")
                .build());
    }
}
```

This is all we need to do. We do not need to override `AuthorizationServerConfigurer`, because the client ID and secret are specified as environment properties.

So, the following should now work:

```
curl first-client:noonewilleverguess@localhost:8080/oauth/token -dgrant_type=password -dscope=any -
dusername=enduser -dpassword=password
```

1.7 How and When to Give Authorization Server an AuthenticationManager

This is a very common question and is not terribly intuitive when `AuthorizationServerEndpointsConfigurer` needs an `AuthenticationManager` instance to be specified. The short answer is: Only when using the Resource Owner Password Flow.

It helps to remember a few fundamentals:

- An `AuthenticationManager` is an abstraction for authenticating users. It typically needs some kind of `UserDetailsService` to be specified in order to be complete.
- End users are specified in a `WebSecurityConfigurerAdapter`.
- OAuth2 Boot, by default, automatically picks up any exposed `AuthenticationManager`.

However, not all flows require an `AuthenticationManager` because not all flows have end users involved. For example, the Client Credentials flow asks for a token based only on the client's authority, not the end user's. And the Refresh Token flow asks for a token based only on the authority of a refresh token.

Also, not all flows specifically require the OAuth 2.0 API itself to have an `AuthenticationManager`, either. For example, the Authorization Code and Implicit flows verify the user when they login (application flow), not when the token (OAuth 2.0 API) is requested.

Only the Resource Owner Password flow returns a code based off of the end user's credentials. This means that the Authorization Server only needs an `AuthenticationManager` when clients are using the Resource Owner Password flow.

The following example shows the Resource Owner Password flow:

```
.authorizedGrantTypes("password", ...)
```

In the preceding flow, your Authorization Server needs an instance of `AuthenticationManager`.

There are a few ways to do this (remember the fundamentals from earlier):

- Leave the OAuth2 Boot defaults (you are not exposing a `AuthorizationServerConfigurer`) and expose a `UserDetailsService`.
- Leave the OAuth2 Boot defaults and expose an `AuthenticationManager`.
- Override `AuthorizationServerConfigurerAdapter` (removing OAuth2 Boot's defaults) and depend on `AuthenticationConfiguration`.
- Override `AuthorizationServerConfigurerAdapter` and manually wire the `AuthenticationManager`.

Exposing a `UserDetailsService`

End users are specified in a `WebSecurityConfigurerAdapter` through a `UserDetailsService`. So, if you use the OAuth2 Boot defaults (meaning you haven't implemented a

`AuthorizationServerConfigurer`), you can expose a `UserDetailsService` and be done, as the following example shows:

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired DataSource dataSource;

    @Bean
    @Override
    public UserDetailsService userDetailsService() {
        return new JdbcUserDetailsManager(this.dataSource);
    }
}
```

Exposing an `AuthenticationManager`

In case you need to do more specialized configuration of the `AuthenticationManager`, you can do so in the `WebSecurityConfigurerAdapter` and then expose it, as the following example shows:

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean(BeansId.AUTHENTICATION_MANAGER)
    @Override
    public AuthenticationManager authenticationManagerBean() {
        return super.authenticationManagerBean();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) {
        auth.authenticationProvider(customAuthenticationProvider());
    }
}
```

If you use the OAuth2 Boot defaults, then it picks up the bean automatically.

Depending on `AuthenticationConfiguration`

Any configured `AuthenticationManager` is available in `AuthenticationConfiguration`. This means that, if you need to have an `AuthorizationServerConfigurer` (in which case you need to do your own autowiring), you can have it depend on `AuthenticationConfiguration` to get the `AuthenticationManager` bean, as the following class shows:

```
@Component
public class CustomAuthorizationServerConfigurer extends
    AuthorizationServerConfigurerAdapter {

    AuthenticationManager authenticationManager;

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

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) {
        // .. your client configuration that allows the password grant
    }

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

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    @Override
    public UserDetailsService userDetailsService() {
        return new MyCustomUserDetailsService();
    }
}
```

Manually Wiring An `AuthenticationManager`

In the most sophisticated case, where the `AuthenticationManager` needs special configuration and you have your own `AuthenticationServerConfigurer`, then you need to both create your own `AuthorizationServerConfigurerAdapter` and your own `WebSecurityConfigurerAdapter`:

```
@Component
public class CustomAuthorizationServerConfigurer extends
    AuthorizationServerConfigurerAdapter {

    AuthenticationManager authenticationManager;

    public CustomAuthorizationServerConfigurer(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) {
        // .. your client configuration that allows the password grant
    }

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

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean(BeansId.AUTHENTICATION_MANAGER)
    @Override
    public AuthenticationManager authenticationManagerBean() {
        return super.authenticationManagerBean();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) {
        auth.authenticationProvider(customAuthenticationProvider());
    }
}
```

1.8 Is Authorization Server Compatible with Spring Security 5.1 Resource Server and Client?

No, not out of the box. Spring Security 5.1 supports only JWT-encoded JWK-signed authorization, and Authorization Server does not ship with a JWK Set URI.

Basic support is possible, though.

In order to configure Authorization Server to be compatible with Spring Security 5.1 Resource Server, for example, you need to do the following:

- Configure it to use JWKs

- Add a JWK Set URI endpoint

Configuring Authorization Server to Use JWKs

To change the format used for access and refresh tokens, you can change out the `AccessTokenConverter` and the `TokenStore`, as the following example shows:

```
@EnableAuthorizationServer
@Configuration
public class JwkSetConfiguration extends AuthorizationServerConfigurerAdapter {

 AuthenticationManager authenticationManager;
 KeyPair keyPair;

 public JwkSetConfiguration(AuthenticationConfiguration authenticationConfiguration,
   KeyPair keyPair) throws Exception {

  this.authenticationManager = authenticationConfiguration.getAuthenticationManager();
  this.keyPair = keyPair;
 }

    // ... client configuration, etc.

 @Override
 public void configure(AuthorizationServerEndpointsConfigurer endpoints) {
  // @formatter:off
  endpoints
   .authenticationManager(this.authenticationManager)
   .accessTokenConverter(accessTokenConverter())
   .tokenStore(tokenStore());
  // @formatter:on
 }

 @Bean
 public TokenStore tokenStore() {
  return new JwtTokenStore(accessTokenConverter());
 }

 @Bean
 public JwtAccessTokenConverter accessTokenConverter() {
  JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
  converter.setKeyPair(this.keyPair);
  return converter;
 }
}
```

Add a JWK Set URI Endpoint

Spring Security OAuth does not support JWKs, nor does `@EnableAuthorizationServer` support adding more OAuth 2.0 API endpoints to its initial set. However, we can add this with only a few lines.

First, you need to add another dependency: `com.nimbusds:nimbus-jose-jwt`. This gives you the appropriate JWK primitives.

Second, instead of using `@EnableAuthorizationServer`, you need to directlyu include its two `@Configuration` classes:

- `AuthorizationServerEndpointsConfiguration`: The `@Configuration` class for configuring the OAuth 2.0 API endpoints, such as what format to use for the tokens.
- `AuthorizationServerSecurityConfiguration`: The `@Configuration` class for the access rules around those endpoints. This is the one that you need to extend, as shown in the following example:

```
@FrameworkEndpoint
class JwkSetEndpoint {
 KeyPair keyPair;

 public JwkSetEndpoint(KeyPair keyPair) {
  this.keyPair = keyPair;
 }

 @GetMapping("/.well-known/jwks.json")
 @ResponseBody
 public Map<String, Object> getKey(Principal principal) {
  RSAPublicKey publicKey = (RSAPublicKey) this.keyPair.getPublic();
  RSAKey key = new RSAKey.Builder(publicKey).build();
  return new JWKSet(key).toJSONObject();
 }
}
```

```
@Configuration
class JwkSetEndpointConfiguration extends AuthorizationServerSecurityConfiguration {
 @Override
 protected void configure(HttpSecurity http) throws Exception {
  super.configure(http);
  http
   .requestMatchers()
    .mvcMatchers("/.well-known/jwks.json")
    .and()
   .authorizeRequests()
    .mvcMatchers("/.well-known/jwks.json").permitAll();
 }
}
```

Then, since you do not need to change `AuthorizationServerEndpointsConfiguration`, you can `@Import` it instead of using `@EnableAuthorizationServer`, as the following example shows:

```
@Import(AuthorizationServerEndpointsConfiguration.class)
@Configuration
public class JwkSetConfiguration extends AuthorizationServerConfigurerAdapter {

    // ... the rest of the configuration from the previous section
}
```

Testing Against Spring Security 5.1 Resource Server

Now you can POST to the `/oauth/token` endpoint (as before) to obtain a token and then present that to a Spring Security 5.1 Resource Server.

2. Resource Server

Spring Security OAuth2 Boot simplifies protecting your resources using Bearer Token authentication in two different token formats: JWT and Opaque.

2.1 Dependencies

To use the auto-configuration features in this library, you need `spring-security-oauth2`, which has the OAuth 2.0 primitives and `spring-security-oauth2-autoconfigure`. Note that you need to specify the version for `spring-security-oauth2-autoconfigure`, since it is not managed by Spring Boot any longer, though it should match Boot's version anyway.

For JWT support, you also need `spring-security-jwt`.

2.2 Minimal OAuth2 Boot Configuration

Creating a minimal Spring Boot resource server consists of three basic steps:

1. Including the dependencies.
2. Including the `@EnableResourceServer` annotation.
3. Specifying a strategy for verifying the bearer token.

Enabling the Resource Server

Similar to other Spring Boot `@Enable` annotations, you can add the `@EnableResourceServer` annotation to the class that contains your `main` method, as the following example shows:

```
@EnableResourceServer
@SpringBootApplication
public class SimpleAuthorizationServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(SimpleAuthorizationServerApplication, args);
    }
}
```

Adding this annotation adds the `OAuth2AuthenticationProcessingFilter`, though it will need one more configuration to know how to appropriately process and validate tokens.

Specifying a Token Verification Strategy

Bearer Tokens typically come in one of two forms: JWT-encoded or opaque. You will need to configure the resource server with one or the other strategy.

JWT

To indicate JWT, simply specify the JWK Set Uri hosted on your Authorization Server:

```
spring:
  security:
    oauth2:
      resource:
        jwk:
          key-set-uri: https://idp.example.com/.well-known/jwks.json
```

Instead of a JWK Set Uri, you can also specify a key.

Note that with this configuration, your authorization server needs to be up in order for Resource Server to start up.

Opaque

To indicate opaque, simply specify the Authorization Server endpoint that knows how to decode the token:

```
spring:
  security:
    oauth2:
      resource:
        token-info-uri: https://idp.example.com/oauth2/introspect
```

> **Note**
>
> It's likely this endpoint requires some kind of authorization separate from the token itself, for example, client authentication.

That's it! But, what do you do with it? We cover that next.

Accessing a Resource

To confirm that Resource Server is correctly processing tokens, you can add a simple controller endpoint like so:

```
@RestController
public class SimpleController
 @GetMapping("/whoami")
 public String whoami(@AuthenticationPrincipal(expression="name") String name) {
  return name;
    }
}
```

Then, obtain an active access token from your Authorization Server and present it to the Resource Server:

```
curl -H "Authorization: $TOKEN" http://localhost:8080/whoami
```

And you should see the value of the `user_name` attribute in the token.

From this point, you may want to learn more about three alternative ways to authenticate using bearer tokens:

- Section 2.3, "How to Use JWT with a Single Key"
- Section 2.4, "How to Configure the Token Info Endpoint"
- Section 2.5, "How to Configure the User Info Endpoint"

2.3 How to Use JWT with a Single Key

Instead of a JWK Set endpoint, you may have a local key you want to configure for verification. While this is weaker due to the key being static, it may be necessary in your situation.

Configuring the resource server with the appropriate symmetric key or PKCS#8 PEM-encoded public key is simple, as can be seen below:

```
spring:
  security:
    oauth2:
      resource:
        jwt:
          key-value: |
            -----BEGIN PUBLIC KEY-----
            MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC...
            -----END PUBLIC KEY-----
```

Tip

The pipe in yaml indicates a multi-line property value.

You can also instead supply a `key-store`, `key-store-password`, `key-alias`, and `key-password` properties.

Or you can use the `key-uri` endpoint to get the key remotely from your authorization server, which is something of a happy medium between static, local configuration and a JWK Set endpoint.

2.4 How to Configure the Token Info Endpoint

The token info endpoint, also sometimes called the introspection endpoint, likely requires some kind of client authentication, either Basic or Bearer. Generally speaking, the bearer token in the `SecurityContext` won't suffice since that is tied to the user. Instead, you'll need to specify credentials that represent this client, like so:

```
spring:
  security:
    oauth2:
      client:
        clientId: client-id
        clientSecret: client-secret
      resource:
        tokenInfoUri: https://idp.example.com/oauth2/check_token
```

By default, this will use Basic authentication, using the configured credentials, to authenticate against the token info endpoint.

2.5 How to Configure the User Info Endpoint

It's atypical for a resource server to need to call a user info endpoint. This is because, fundamentally, a resource server is about authorizing a request, not authenticating it. That said, it is at times necessary.

If you specify a user info endpoint like so:

```
spring:
  security:
    oauth2:
      resource:
        userInfoUri: https://idp.example.com/oauth2/userinfo
```

Then Resource Server will send it the bearer token that is part of the request and enhance the `Authentication` object with the result.

Customizing the User Info Request

Internally, Resource Server uses an `OAuth2RestTemplate` to invoke the `/userinfo` endpoint. At times, it may be necessary to add filters or perform other customization for this invocation. To customize the creation of this bean, you can expose a `UserInfoRestTemplateCustomizer`, like so:

```
@Bean
public UserInfoRestTemplateCustomizer customHeader() {
 return restTemplate ->
   restTemplate.getInterceptors().add(new MyCustomInterceptor());
}
```

This bean will be handed to a `UserInfoTemplateFactory` which will add other configurations helpful to coordinating with the `/userinfo` endpoint.

And, of course, you can replace the `UserInfoTemplateFactory` completely, if you need complete control over `OAuth2RestTemplate's configuration.

2.6 Customizing Authorization Rules

Similar to how Spring Security works, you can customize authorization rules by endpoint in Spring Security OAuth, like so:

```
public class HasAuthorityConfig
  extends ResourceServerConfigurerAdapter {

 @Override
 public void configure(HttpSecurity http) throws Exception {
  // @formatter:off
  http
   .authorizeRequests()
    .antMatchers("/flights/**").hasAuthority("#oauth2.hasScope('message:read')")
    .anyRequest().authenticated();
  // @formatter:on
 }
```

Though, note that if a server is configured both as a resource server and as an authorization server, then there are certain endpoint that require special handling. To avoid configuring over the top of those endpoints (like `/token`), it would be better to isolate your resource server endpoints to a targeted directory like so:

```
public class ResourceServerEndpointConfig
  extends ResourceServerConfigurerAdapter {

 @Override
 public void configure(HttpSecurity http) throws Exception {
  // @formatter:off
  http
   .antMatchers("/resourceA/**", "/resourceB/**")
   .authorizeRequests()
    .antMatchers("/resourceA/**").hasAuthority("#oauth2.hasScope('resourceA:read')")
    .antMatchers("/resourceB/**").hasAuthority("#oauth2.hasScope('resourceB:read')")
    .anyRequest().authenticated();
  // @formatter:on
 }
```

As the above configuration will target your resource endpoints and not affect authorization server-specific endpoints.

2.7 Less Common Features

Changing the Token Type

Google and certain other third-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. However, if you need to change it, you can set `security.oauth2.resource.token-type`.

Changing the Filter Order

OAuth2 resources are protected by a filter chain with the order specified by `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();
    }
}
```

Permitting the /error Endpoint

Resource Server, when also configured as a client, may rely on a request-scoped `OAuth2ClientContext` bean during the authentication process. And, in some error situations, Resource Server forwards to the ERROR servlet dispatcher.

By default, request-scoped beans aren't available in the ERROR dispatch. And, because of this, you may see a complaint about the `OAuth2ClientContext` bean not being available.

The simplest approach may be to permit the `/error` endpoint, so that Resource Server doesn't try and authenticate the request:

```
public class PermitErrorConfig extends ResourceServerConfigurerAdapter {
    @Override
 public void configure(HttpSecurity http) throws Exception {
  // @formatter:off
  http
   .authorizeRequests()
    .antMatchers("/error").permitAll()
    .anyRequest().authenticated();
  // @formatter:on
 }
}
```

Other solutions are to configure Spring so that the `RequestContextFilter` is registered with the error dispatch or to register a `RequestContextListener` bean.

3. Client

To make your web application into an OAuth2 client, you can add `@EnableOAuth2Client` and Spring Boot creates an `OAuth2ClientContext` and `OAuth2ProtectedResourceDetails` that are necessary to create an `OAuth2RestOperations`. Spring Boot does not automatically create such a bean, but you can easily create your own, as the following example shows:

```
@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). However, in addition, it needs to know the authorization and token URIs in the Authorization Server, as the following example shows:

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 redirects to Github for authorization when you attempt to use the `OAuth2RestTemplate`. If you are already signed into Github. you should not even notice that it has authenticated. These specific credentials work only if your application is running on port 8080 (you can register your own client application 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 does not like header authentication). In fact, the `security.oauth2.client.*` properties are bound to an instance of `AuthorizationCodeResourceDetails`, so all of 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, you are asking

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

4. Single Sign On

You can use an OAuth2 Client 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 (described earlier) supports this through 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 shown in the preceding section can protect all its resources and authenticate by 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 earlier):

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

Example 4.1 application.yml

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 s(o you can add a “home” page for instance,) you can add `@EnableOAuth2Sso` to a `WebSecurityConfigurerAdapter`. The annotation causes it to be decorated and enhanced with the necessary pieces to get the `/login` path working. In the following example, 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 `/error` endpoint. This means that, if there is some problem during Single Sign On that requires the application to redirect to the `/error` page, 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`, as the following example shows:

```
@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

You can specify various properties inside your `application.properties` or `application.yml` files 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 legitimate to define your own properties.

> **Warning**
>
> This sample file is meant as a guide only. Do **not** copy and 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
```