Lesson 1: The New OAuth2 Client Support (NEW - Text Only)

1. Goal

In this lesson, we'll implement the Client as part of the simple OAuth application we're building here.

We're going to use the new OAuth stack here as we now have full Client support rolled out.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m18-oauth-client-start.

If you want to skip and see the complete implementation, feel free to jump ahead and import: m18-oauth-client.

We'll also be running the other two applications: the Authorization and the Resource Servers. These aren't the focus of this lesson, but we naturally do need them running as we're developing the Client.

You'll also find a Client implementation using the old stack in the lesson code, if you need that as a reference.

Now let's start working on the new Client here.

Configuration

We're starting from a simple Boot application that's just an empty application.

Let's add the new OAuth client dependency:

```HTML
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-client</artifactId>
</dependency>
```

Now that we have the support in, we're going to drive the core of the Client functionality via properties.

Let's start by defining a custom client:

```HTML
spring:
  security:
    oauth2:
      client:
        registration:
          custom:
            client-id: lssClient
            client-secret: lssSecret
            scope: read,write,user
            authorization-grant-type: authorization_code
            redirect-uri: http://localhost:8082/um-webapp-client/login/oauth2/code/custom
```

Here, we've used the *spring.security.oauth2.client.registration* namespace, which is the root namespace for registering any Client.

We've defined the Client exactly as it's registered in the Authorization Server, by setting the *client-id* and *client-secret* properties, as well as the scopes and authorization grant type.

Finally, we set the redirect URI property to complete our Client definition.

Next, we need to define the Service Provider for our Client which is basically the information of the Authorization Server. This is using the *provider* namespace and it's, naturally, also *custom*.

What we need here is to point to the URIs that will be involved in the process.

Keep in mind our Authorization Server is running on:

http://localhost:8083/um-webapp-auth-server/

And we'll just use the standard URIs:

authorization-uri: /oauth/authorize

token-uri: /oauth/token

So we'll add the following properties in our *application.yml*:

```HTML
spring:
  security:
    oauth2:
      client:
        provider:
          custom:
            authorization-uri: http://localhost:8083/um-webapp-auth-server/oauth/authorize
            token-uri: http://localhost:8083/um-webapp-auth-server/oauth/token
```

We also need to define a user info endpoint. Our Resource Server is running on:

http://localhost:8081/um-webapp-resource-server/

And again, we'll use the standard URI for this, which is */users/info*. So let's add this to our configuration file:

```HTML
spring:
  security:
    oauth2:
      client:
        provider:
          custom:
            authorization-uri: http://localhost:8083/um-webapp-auth-server/oauth/authorize
            token-uri: http://localhost:8083/um-webapp-auth-server/oauth/token
            user-info-uri: http://localhost:8081/um-webapp-resource-server/users/info
            user-name-attribute: user_name
```

Finally, we do need to specify exactly what the name of the user attribute is, which is simply *user_name*.

Now, let's enable the new functionality in our *SecurityConfig*:

```JAVA
```

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
        //...
            .oauth2Login()
            .and()
            .logout();
    }
}
```

RestTemplate Client

The goal of this Client application is to be able to consume the Resource Server on behalf of the User.

So what we'll need here is a simple HTTP client. But, instead of using any plain HTTP client and then having to deal with sending the token manually, we'll make use of the framework support.

We'll define a *RestTemplate* bean that is already set up and configured to send the Access Token when consuming the Resource Server:

JAVA

```
@Bean
public RestTemplate restTemplate() {
    RestTemplate restTemplate = new RestTemplate();
    return restTemplate;
}
```

This is the simple bean but, clearly, we need to make it OAuth aware.

We're going to do that by using an interceptor which the *RestTemplate* has native support for, called *AuthorizationHeaderInterceptor*:

JAVA

```
@Bean
public RestTemplate restTemplate(OAuth2AuthorizedClientService clientService) {
    RestTemplate restTemplate = new RestTemplate();

    List<ClientHttpRequestInterceptor> interceptors = restTemplate.getInterceptors();
    if (CollectionUtils.isEmpty(interceptors)) {
        interceptors = new ArrayList<>();
    }
    interceptors.add(new AuthorizationHeaderInterceptor(clientService));
    restTemplate.setInterceptors(interceptors);
    return restTemplate;
}
```

This interceptor is build exactly for this purpose: to add the Autorization Bearer token to the request.

It also takes in the Client Service bean, which we've injected here as a simple parameter.

Client Controller

Let's now implement our own controller in the Client application, which will simply consume the Resource Server.

What's interesting here is that even though the Resource Server is a REST API, the Client application can chose to expose the data however it needs to.

For example, we could also expose data as a REST API and have a more JS-centric front-end app here. Or, we could go with an entirely different style since the Client application is a fully independent app.

It can, for instance, go with a more traditional MVC-style implementation. So let's do that here. We'll inject the *RestTemplate* into our Controller and implement the *get users* operation:

```java
@Controller
public class UserClientController {

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping("/users")
    public String getUsers(Model model) {
        List<User> users = restTemplate.getForObject("http://localhost:8081/um-webapp-resource-server/api/users/", List.class);
        model.addAttribute("users", users);
        return "users";
    }
}
```

We can, of course, add more operations to this Controller.

Now we can access this in the browser:

http://localhost:8082/um-webapp-client/users

3. Resources

- Using Spring Security 5 to integrate with OAuth 2-secured services

- Spring Security OAuth2 Client

Lesson 3: The New Resource Server Support (NEW - Text Only)

1. Goal

In this lesson, we''re going to build out a simple Resource Server - using the new OAuth stack in Spring Security 5.

2. Lesson Notes

The relevant module you need to import for this lesson is: m18-oauth-resource-server.

Maven Dependencies

First, we need to add the following dependencies to our *pom.xml*:

```HTML
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-jose</artifactId>
</dependency>
```

Here, we've added the 2 additional dependencies which are new in Spring 5 and Spring Boot 2.

The spring-security-oauth2-resource-server is the one that contains the new Resource Server support.

The spring-security-oauth2-jose library contains support for the JOSE framework, which includes JWT and JWK.

Configuration

Next, we'll secure our API using the new Resource Server options.

For this, we'll add the Resource Server support using the *oauth2ResourceServer()* DSL to our standard Spring Security configuration class:

```JAVA
http.authorizeRequests()
  .anyRequest().authenticated()
  .and()
  .oauth2ResourceServer()
  .jwt();
```

We can notice how simple the configuration is compared to the previous one. The *oauth2ResourceServer()* method replaces the previously used annotation *@EnableResourceServer* which is no longer available.

We're also using the *jwt()* method to configure our Resource Server to use JWT tokens to validate authentication.

Next, let's add access rules for URLs in our application:

JAVA

```
http.authorizeRequests()
  .antMatchers(HttpMethod.GET,"/users/**","/api/foos/**").hasAuthority("SCOPE_read")
  .antMatchers(HttpMethod.POST,"/api/foos/**").hasAuthority("SCOPE_write")
  .anyRequest().authenticated()
  .and()
  .oauth2ResourceServer()
  .jwt();
```

Here we're using values of the form SCOPE_ to restrict the url to JWT that have the specified scopes. This is similar to the existing ROLE_ authorities.

Finally, we need to set the *jwk-set-uri* property in our *application.properties*:

HTML

```
spring.security.oauth2.resourceserver.jwt.jwk-set-uri=http://localhost:8083/um-webapp-auth-server/endpoint/jwks.json
```

Using this configuration means the Resource Server doesn't need to ping the Authorization Server for validating the tokens. It will do so based on the JWK set specified.

Alternatively, if the Authorization Server supports the Provider Configuration endpoint, then we can configure the *issuer-uri*:

HTML

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=https://idp.example.com
```

This informs the Resource Server which Authorization Server URI to use for validating tokens.

Finally, note that we're only focusing on JWT minimally, because we'll explore JWT in a future lesson in more detail.

Also, note that **this lesson is focused on the Resource Server only.**

The Authorization Server and the Client are covered by other lessons in this module.

Here, all we need to do is have them running, but the details of their implementation isn't really important for our Resource Server.

3. Resources

- Spring Security Java Configuration Reference Manual