Reactive Spring Security 5 Hands-On Workshop

Andreas Falk

Table of Contents

Chapter 1. Introduction

Target of this workshop is to learn how to make an initially unsecured (reactive) web application more and more secure step-by-step.

This will be done in following steps:

1. Add spring boot security starter dependency for simple auto configuration of security
2. Customize authentication configuration (provide our own user store)
3. Add authorization (access controls) to web and method layers
4. Implement automated security integration tests
5. Experiment with new OAuth2 Login Client and Resource Server of Spring Security 5.1

1.1. Requirements for this workshop

- Git
- A Java 8 or 9 JDK
- Any Java IDE capable of building with Gradle (IntelliJ, Eclipse, VS Code, …)
- Basic knowledge of Reactive Systems and reactive programming using Spring WebFlux & Reactor

As we are building the samples using Gradle your Java IDE should be capable use this. As IntelliJ user support for Gradle is included by default. As an Eclipse user you have to install a plugin via the marketplace

To get the workshop project you either can just clone the repository using

```
https://github.com/andifalk/reactive-spring-security-5-workshop.git
```

or

```
git@github.com:andifalk/reactive-spring-security-5-workshop.git
```

or simply download it as a zip archive.

After that you can import the workshop project into your IDE

- IntelliJ: "New project from existing sources..."
- Eclipse: "Import/Gradle/Existing gradle project"
- Visual Studio Code: Just open the corresponding project directory

Chapter 2. Common Web Security Risks

In this workshop you will strive various parts of securing a web application that fit into the OWASP Top 10 2017 list.

We will look at:

- A2: Broken Authentication

- A3: Sensitive Data Exposure

- A5: Broken Access Control

- A6: Security Misconfiguration

- A10: Insufficient Logging & Monitoring

OWASP Top 10 - 2013		OWASP Top 10 - 2017
A1 – Injection	➜	A1:2017-Injection
A2 – Broken Authentication and Session Management	➜	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	➜	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

You may also have a look into the OWASP ProActive Controls document which describes how to develop your applications using good security patterns.

> ℹ️ You will find more sources of information about security referenced in the appendix section.

Chapter 3. The workshop application

In this workshop you will be provided a finished but completely unsecured reactive web application. This library server application provides a RESTful service for administering books and users.

The RESTful service for books is build using the Spring WebFlux annotation model and the RESTful service for users is build using the Spring WebFlux router model.

The application contains a complete documentation for the RESTful API build with spring rest docs which you can find in the directory *build/asciidoc/html5* after performing a full gradle build.

The domain model of this application is quite simple and just consists of *Book* and *User*. The packages of the application are organized as follows:

- **api**: Contains the complete RESTful service
- **business**: All the service classes (quite simple for workshop, usually containing business logic)
- **dataaccess**: All domain models and repositories
- **config**: All spring configuration classes

> For users of IntelliJ you find http scripts to test all the RESTful services in sub directory *http* of all projects.

There are three target user roles for this application:

- **Standard users**: A standard user can borrow and return his currently borrowed books
- **Curators**: A curator user can add or delete books
- **Administrators**: An administrator user can add or remove users

The application is build using

- Spring 5 WebFlux on Netty
- Spring Data MongoDB with reactive driver
- In-memory Mongodb to have an easier setup for the workshop

The following subsections give a very condensed introduction to the basics of Reactive Systems, the Project Reactor and Spring WebFlux.
This might help to better understand the sample application for beginners of Reactive.

3.1. Reactive Systems & Streams

> Reactive Systems are Responsive, Resilient, Elastic and Message Driven (Asynchronous).
>
> — https://www.reactivemanifesto.org

- **Responsiveness** means the system responds in a timely manner and is the cornerstone of usability
- **Resilience** means the system stays responsive in the face of failure
- **Elasticity** means that the throughput of a system scales up or down automatically to meet varying demand as resource is proportionally added or removed
- **Message Driven** systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling

> Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure..

— http://www.reactive-streams.org/

- **Back-Pressure**: When one component is struggling to keep-up, the system as a whole needs to respond in a sensible way. Back-pressure is an important feedback mechanism that allows systems to gracefully respond to load rather than collapse under it.

3.1.1. Project Reactor

The project Reactor is a Reactive library for building non-blocking applications on the JVM based on the Reactive Streams Specification and can help to build Reactive Systems.

Reactor is a fully non-blocking foundation and offers backpressure-ready network engines for HTTP (including Websockets), TCP and UDP.

Reactor introduces composable reactive types that implement Publisher but also provide a rich vocabulary of operators, most notably *Flux* and *Mono*.

A *Mono<T>* is a specialized *Publisher<T>* that emits at most one item and then optionally terminates with an onComplete signal or an onError signal.

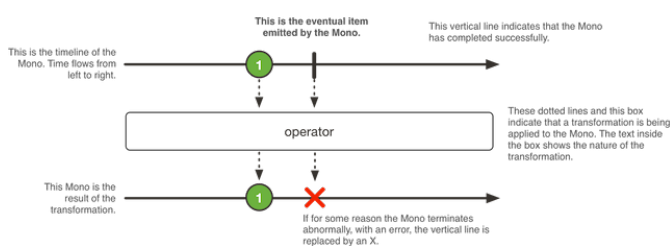*Figure 1. Mono, an Asynchronous 0-1 Result (source: projectreactor.io)*

A *Flux<T>* is a standard *Publisher<T>* representing an asynchronous sequence of 0 to N emitted items, optionally terminated by either a completion signal or an error.
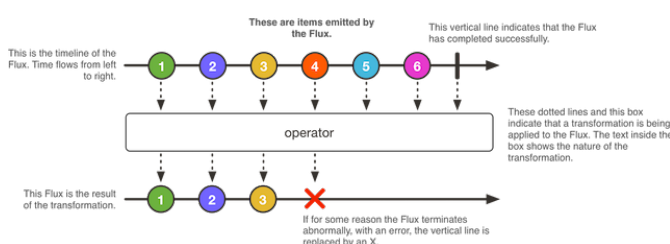

Figure 2. Flux, an Asynchronous Sequence of 0-N Items (source: projectreactor.io)

3.1.2. Spring WebFlux

Spring WebFlux was added in Spring Framework 5.0. It is fully non-blocking, supports Reactive Streams back pressure, and runs on servers such as Netty, Undertow, and Servlet 3.1+ containers.

Spring Webflux depends on Reactor and uses it internally to compose asynchronous logic and to provide Reactive Streams support.
It provides two programming models:

- Annotated Controllers: This is uses the same annotations as in the Spring MVC part

- Functional Endpoints: This provides a lambda-based, lightweight, functional programming model

Chapter 4. Workshop Organization

This interactive hands-on workshop is organized into several step with one step building upon each other. There is a separate project for each step:

- 00-library-server: This contains the initial unsecured application
- 01-library-server: This has just added the spring boot starter dependencies for spring security
- 02-library-server: This adds a persistent user store for authentication
- 03-library-server: This adds custom authentication and authorization rules
- 04-library-server: This adds automatic integration tests for authorization
- 05-oauth2: This adds an OAuth2 login clientand resource server for interacting with an authorization server for login

Chapter 5. Workshop Steps

To start the workshop please begin by adapting the *00-library-server* application.

> ℹ️ If you are not able to keep up with completing a particular step you always can just start over with the existing application of next step.
>
> For example if you could not manage to complete the tutorial based on *01-library-server* just continue using *02-library-server*.

5.1. Step 1: Auto Configuration

In the first step we start quite easy by just adding the spring boot starter dependency for spring security.

We just need to add the following two dependencies to the *build.gradle* file of the initial application (*00-library-server*).
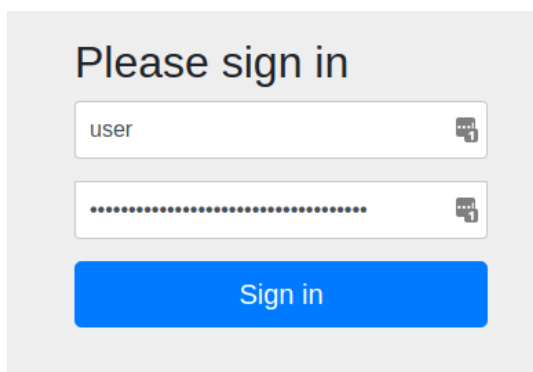
build.gradle

```
dependencies {
    ...
    compile('org.springframework.boot:spring-boot-starter-security')  ①
    ...
    testCompile('org.springframework.security:spring-security-test')  ②
}
```

① The spring boot starter for spring security

② Adds testing support for spring security

Please start the application by running the class *LibraryServerApplication*.

If you browse to localhost:8080/books then you will notice that a login form appears in the browser window.

But wait - what are the credentials for a user to log in?

With spring security autoconfigured by spring boot the credentials are as follows:

- Username=user

- Password=<Look into the console log!>

console log

```
INFO 18465 --- [  restartedMain] ctiveUserDetailsServiceAutoConfiguration :

Using default security password: ded10c78-0b2f-4ae8-89fe-c267f9a29e1d
```

As you can see, if Spring Security is on the classpath, then the web application is secured by default. Spring boot auto-configured basic authentication and form based authentication for all web endpoints.

This also applies to all actuator endpoints like /actuator/health. All monitoring web endpoints can now only be accessed with an authenticated user. See Actuator Security for details.

Additionally spring security improved the security of the web application automatically for:

- Session Fixation: Session Fixation is an attack that permits an attacker to hijack a valid user session.
  If you want to learn more about this please read the Session Fixation page at OWASP

- Cross Site Request Forgery (CSRF): Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.
  If you want to know what CSRF really is and how to mitigate this attack please consult CSRF attack description at OWASP

- Default Security Headers: This automatically adds all recommended security response headers to all http responses. You can find more information about this topic in the OWASP Secure Headers Project

default security response headers

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

Spring security 5 also added a bit more user friendly logout functionality out of the box. If you direct your browser to localhost:8080/logout you will see the following dialog on the screen.

Are you sure you want to log out?

Log Out

This concludes the first step.

> ℹ️ You find the completed code in project *01-library-server*.

Now let's proceed to next step and start with customizing the authentication part.

5.2. Step 2: Customize Authentication

Now it is time to start customizing the auto-configuration.

The spring boot auto-configuration will back-off a bit in this step and will back-off completely in next step.

We start by replacing the default user/password with our own persistent user storage (already present in MongoDB). To do this we add a new class *WebSecurityConfiguration* to package *com.example.library.server.config* having the following contents.

WebSecurityConfiguration class

```
package com.example.library.server.config;

import org.springframework.context.annotation.Bean;
import
org.springframework.security.config.annotation.web.reactive.EnableWebFluxSecurity;
import org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;

@EnableWebFluxSecurity ①
public class WebSecurityConfiguration {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder(); ②
    }

}
```

① This auto-configures the SecurityWebFilterChain

② This adds the new delegating password encoder introduced in spring security 5

The *WebSecurityConfiguration* implementation does two important things:

1. This adds the *SecurityWebFilterChain*. If you already have secured servlet based spring mvc web

applications then you might know what's called the *spring security filter chain*. In spring
webflux the *SecurityWebFilterChain* is the similar approach based on matching a request with
one or more WebFilter.

2. Configures a *PasswordEncoder*. A password encoder is used by spring security to encrypt
   passwords and to check if a given password matches the encrypted one.

PasswordEncoder interface

```
package org.springframework.security.crypto.password;

public interface PasswordEncoder {

    String encode(CharSequence rawPassword); ①

    boolean matches(CharSequence rawPassword, String encodedPassword); ②
}
```

① Encrypts the given cleartext password

② Validates the given cleartext password with the encrypted one (without revealing the
  unencrypted one)

In spring security 5 creating an instance of the *DelegatingPasswordEncoder* is much easier by using
the class *PasswordEncoderFactories*.

DelegatingPasswordEncoder class

```
package org.springframework.security.crypto.factory;

public class PasswordEncoderFactories {
    ...
    public static PasswordEncoder createDelegatingPasswordEncoder() {
        String encodingId = "bcrypt"; ①
        Map<String, PasswordEncoder> encoders = new HashMap<>();
        encoders.put(encodingId, new BCryptPasswordEncoder()); ②
        encoders.put("ldap", new LdapShaPasswordEncoder());
        encoders.put("MD4", new Md4PasswordEncoder());
        encoders.put("MD5", new MessageDigestPasswordEncoder("MD5"));
        encoders.put("noop", NoOpPasswordEncoder.getInstance());
        encoders.put("pbkdf2", new Pbkdf2PasswordEncoder());
        encoders.put("scrypt", new SCryptPasswordEncoder());
        encoders.put("SHA-1", new MessageDigestPasswordEncoder("SHA-1"));
        encoders.put("SHA-256", new MessageDigestPasswordEncoder("SHA-256"));
        encoders.put("sha256", new StandardPasswordEncoder());

        return new DelegatingPasswordEncoder(encodingId, encoders);
    }
    ...
}
```

① BCrypt is the default for encrypting passwords

② Suitable encoders for decrypting are selected based on prefix in encrypted value

To have encrypted passwords in our MongoDB store we need to tweak our existing *DataInitializer* a bit with the *PasswordEncoder* we just have configured.

DataInitializer class

```
package com.example.library.server;
...
import org.springframework.security.crypto.password.PasswordEncoder;
...

@Component
public class DataInitializer implements CommandLineRunner {

    ...
    private final PasswordEncoder passwordEncoder; ①

    @Autowired
    public DataInitializer(BookRepository bookRepository, UserRepository
userRepository,
                           IdGenerator idGenerator, PasswordEncoder passwordEncoder)
{
        ...
        this.passwordEncoder = passwordEncoder;
    }

    ...

    private void createUsers() {
        ...
        userRepository
                .save(
                        new User(
                                USER_IDENTIFIER,
                                "user@example.com",
                                passwordEncoder.encode("user"), ②
                                "Library",
                                "User",
                                Collections.singletonList(Role.USER)))
                .subscribe();
        ...
    }
    ...
}
```

① Inject *PasswordEncoder* to encrypt user passwords

② Change cleartext passwords into encrypted ones (using BCrypt as default)

Now that we already have configured the encrypting part for passwords of our user storage we

need to connect our own user store (the users already stored in the MongoDB) with spring security's authentication manager.

This is done in two steps:

In the first step we need to implement spring security's definition of a user called *UserDetails*.

LibraryUser class

```java
package com.example.library.server.security;

import com.example.library.server.business.UserResource;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.userdetails.UserDetails;

import java.util.Collection;
import java.util.stream.Collectors;

public class LibraryUser implements UserDetails { ①

    private final UserResource userResource; ②

    public LibraryUser(UserResource userResource) {
        this.userResource = userResource;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return AuthorityUtils
                .commaSeparatedStringToAuthorityList(
                        userResource.getRoles().stream()
                                .map(rn -> "ROLE_" + rn.name())
                                .collect(Collectors.joining(",")));
    }

    @Override
    public String getPassword() {
        return userResource.getPassword();
    }

    @Override
    public String getUsername() {
        return userResource.getEmail();
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }
```

```java
    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }

    public UserResource getUserResource() {
        return userResource;
    }
}
```

① To provide our own user store we have to implement the spring security's predefined interface *UserDetails*

② The implementation for *UserDetails* is backed up by our existing *UserResource* value object

In the second step we need to implement spring security's interface *ReactiveUserDetailsService* to integrate our user store with the authentication manager.

LibraryReactiveUserDetailsService class

```
package com.example.library.server.security;

import com.example.library.server.business.UserService;
import org.springframework.security.core.userdetails.ReactiveUserDetailsService;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;

@Service
public class LibraryReactiveUserDetailsService implements ReactiveUserDetailsService {
①

    private final UserService userService; ②

    public LibraryReactiveUserDetailsService(UserService userService) {
        this.userService = userService;
    }

    @Override
    public Mono<UserDetails> findByUsername(String username) { ③
        return userService.findOneByEmail(username).map(LibraryUser::new);
    }
}
```

① To provide our own user store we have to implement the spring security's predefined interface *ReactiveUserDetailsService* which is the binding component between the authentication service and our *LibraryUser*

② To search and load the targeted user for authentication we use our existing *UserService*

③ This will be called when authentication happens to get user details for validating the password and adding this user to the security context

After completing this part of the workshop we now still have the auto-configured *SecurityWebFilterChain* but we have replaced the default user with our own users from our MongoDB persistent storage.

If you restart the application now you have to use the following user credentials to log in:

Table 1. User credentials

User	Password	Roles
user@example.com	user	USER
curator@example.com	curator	USER, CURATOR
admin@example.com	admin	USER, CURATOR, ADMIN

This is the end of step 2 of the workshop.

You find the completed code in project *02-library-server*.

In the next workshop part we also adapt the *SecurityWebFilterChain* to our needs and add authorization rules (in web and method layer) for our application.

5.3. Step 3: Add Authorization

In this part of the workshop we want to add our customized authorization rules for our application.

As a result of the previous workshop steps we now have authentication for all our web endpoints (including the actuator endpoints) and we can log in using our own users. But here security does not stop.

We know who is using our application (**authentication**) but we do not have control over what this user is allowed to do in our application (**authorization**).

As a best practice the authorization should always be implemented on different layers like the web and method layer. This way the authorization still prohibits access even if a user manages to bypass the web url based authorization filter by playing around with manipulated URL's.

Our required authorization rule matrix looks like this:

Table 2. Authorization rules for library-server

URL	Http method	Restricted	Roles with access
/**.css,**/.jpg,/*.ico,...	All	No	—
/books	POST	Yes	CURATOR
/books	DELETE	Yes	CURATOR
/users	All	Yes	ADMIN
/actuator/health	GET	No	—
/actuator/info	GET	No	—
/actuator/*	GET	Yes	ADMIN
/*	All	Yes	All authenticated ones

All the web layer authorization rules are configured in the *WebSecurityConfiguration* class by adding a new bean for *SecurityWebFilterChain*. Here we also already switch on the support for method layer authorization by adding the annotation *@EnableReactiveMethodSecurity*.

WebSecurityConfiguration class

```
package com.example.library.server.config;

...

@EnableWebFluxSecurity
@EnableReactiveMethodSecurity ①
public class WebSecurityConfiguration {

    @Bean ②
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        return http
                .authorizeExchange()

.matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll() ③
                .matchers(EndpointRequest.to("health")).permitAll() ④
                .matchers(EndpointRequest.to("info")).permitAll()
                .matchers(EndpointRequest.toAnyEndpoint()).hasRole(Role.ADMIN.name())
⑤
                .pathMatchers(HttpMethod.POST, "/books").hasRole(Role.CURATOR.name())
⑥
                .pathMatchers(HttpMethod.DELETE,
"/books").hasRole(Role.CURATOR.name())
                .pathMatchers("/users/**").hasRole(Role.ADMIN.name()) ⑦
                .anyExchange().authenticated() ⑧
                .and()
                .httpBasic().and().formLogin().and() ⑨
                .logout().logoutSuccessHandler(logoutSuccessHandler()) ⑩
                .and()
                .build();
    }

    @Bean ⑪
    public ServerLogoutSuccessHandler logoutSuccessHandler() {
        RedirectServerLogoutSuccessHandler logoutSuccessHandler = new
RedirectServerLogoutSuccessHandler();
        logoutSuccessHandler.setLogoutSuccessUrl(URI.create("/books"));
        return logoutSuccessHandler;
    }

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

}
```

① This adds support for method level authorization

② Configures authentication and web layer authorization for all URL's of our REST api

③ All static resources (favicon.ico, css, images, …) can be accessed without authentication

④ Actuator endpoints for *health* and *info* can be accessed without authentication

⑤ All other actuator endpoints require authentication

⑥ Modifying access to books require authenticated user having the 'CURATOR' role

⑦ Access to users require authenticated user having the 'ADMIN' role

⑧ All other web endpoints require authentication

⑨ Authentication can be performed using basic authentication or form based login

⑩ After logging out it redirects to URL configured in the logout success handler

⑪ The configured login success handler redirects to /books resource

We also add a a *ServerLogoutSuccessHandler* bean to redirect back to the */books* endpoint after a logout to omit the error message we got so far by redirecting to a non-existing page.

We continue with authorization on the method layer by adding the rules to our business service classes *BookService* and *UserService*. To achieve this we use the *@PreAuthorize* annotations provided by spring security. Same as other spring annotations (e.g. @Transactional) you can put *@PreAuthorize* annotations on global class level or on method level.

Depending on your authorization model you may use *@PreAuthorize* to authorize using static roles or to authorize using dynamic expressions (usually if you have roles with permissions).
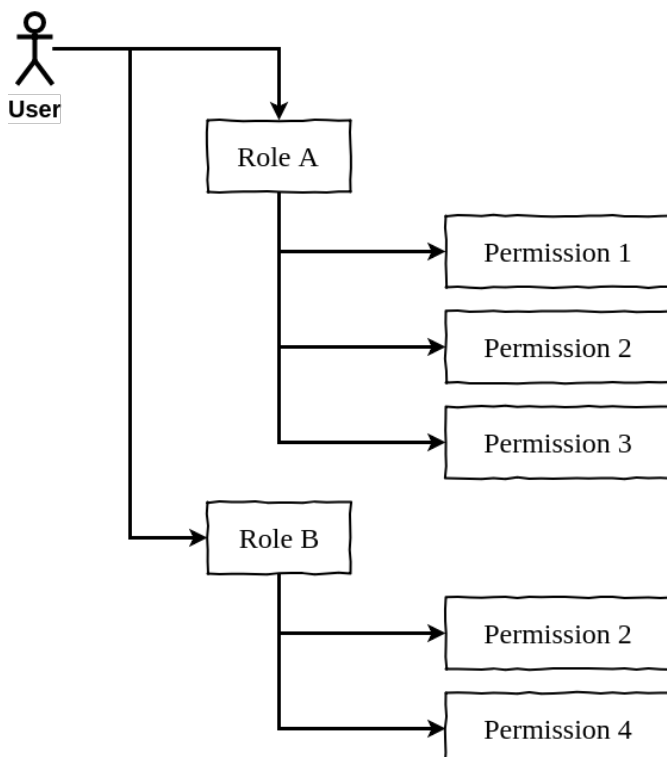

Figure 3. Roles and Permissions

If you want to have a permission based authorization you can use the predefined interface *PermissionEvaluator* inside the *@PreAuthorize* annotations like this:

```
class MyService {
    @PreAuthorize("hasPermission(#uuid, 'user', 'write')")
    void myOperation(UUID uuid) {...}
}
```

PermissionEvaluator class

```
package org.springframework.security.access;

...
public interface PermissionEvaluator extends AopInfrastructureBean {

    boolean hasPermission(Authentication authentication, Object targetDomainObject,
            Object permission);

    boolean hasPermission(Authentication authentication, Serializable targetId,
            String targetType, Object permission);
}
```

In the workshop due to time constraints we have to keep things simple so we just use static roles. Here it is done for the all operations of the book service.

BookService class

```
package com.example.library.server.business;

...
import org.springframework.security.access.prepost.PreAuthorize;
...

@Service
@PreAuthorize("hasAnyRole('USER', 'CURATOR', 'ADMIN')") ①
public class BookService {

    ...

    @PreAuthorize("hasRole('CURATOR')") ②
    public Mono<Void> create(Mono<BookResource> bookResource) {
        return bookRepository.insert(bookResource.map(this::convert)).then();
    }

    ...

    @PreAuthorize("hasRole('CURATOR')") ③
    public Mono<Void> deleteById(UUID uuid) {
        return bookRepository.deleteById(uuid).then();
    }
    ...
}
```

① In general all users (having either of these 3 roles) can access RESTful services for books

② Only users having role 'CURATOR' can access this RESTful service to create books

③ Only users having role 'CURATOR' can access this RESTful service to delete books

And now we add it the same way for the all operations of the user service.

UserService class

```
package com.example.library.server.business;
...
import org.springframework.security.access.prepost.PreAuthorize;
...
@Service
@PreAuthorize("hasRole('ADMIN')") ①
public class UserService {

    ...

    @PreAuthorize("isAnonymous() or isAuthenticated()") ②
    public Mono<UserResource> findOneByEmail(String email) {
        return userRepository.findOneByEmail(email).map(this::convert);
    }

    ...
}
```

① In general only users having role 'ADMIN' can access RESTful services for users

② As this operation is used by the *LibraryUserDetailsService* to perform authentication this has to be accessible for anonymous users (unless authentication is finished successfully anonymous users are unauthenticated users)

Now that we have the current user context available in our application we can use this to automatically set this user as the one who has borrowed a book or returns his borrowed book. The current user can always be evaluated using the *ReactiveSecurityContextHolder* class. But a more elegant way is to just let the framework put the current user directly into our operation via *@AuthenticationPrincipal* annotation.

BookRestController class

```
package com.example.library.server.api;

import org.springframework.security.core.annotation.AuthenticationPrincipal;

@RestController
public class BookRestController {

    ...

    @PostMapping("/books/" + PATH_BOOK_ID + "/borrow")
    public Mono<Void> borrowBookById(
            @PathVariable(PATH_VARIABLE_BOOK_ID) UUID bookId, @AuthenticationPrincipal
LibraryUser user) { ①
        return bookService.borrowById(bookId, user.getUserResource().getId());
    }

    @PostMapping("/books/" + PATH_BOOK_ID + "/return")
    public Mono<Void> returnBookById(@PathVariable(
            PATH_VARIABLE_BOOK_ID) UUID bookId, @AuthenticationPrincipal LibraryUser
user) { ②
        return bookService.returnById(bookId, user.getUserResource().getId());
    }

    ...
}
```

① Now that we have an authenticated user context we can add the current user as the one to borrow a book

② Now that we have an authenticated user context we can add the current user as the one to return his borrowed a book

So please go ahead and re-start the application and try to borrow a book with an authenticated user. If you are an IntelliJ user you can use the provided *book.http* file in subdirectory *http*.

At first you will notice that even with the correct basic authentication header you get an error message like this one:

CSRF error output

```
POST http://localhost:8080/books

HTTP/1.1 403 Forbidden
transfer-encoding: chunked
Content-Type: text/plain
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1 ; mode=block

CSRF Token has been associated to this client

Response code: 403 (Forbidden)
```

The library-server expects a CSRF token in the request but did not find one. If you use common UI frameworks like Thymeleaf or JSF (on the serverside) or a clientside one like Angular then these already handle this CSRF processing.

In our case we do not have such handler. To successfully tra the borrow book request you have to switch off CSRF in the library server.
This is done like this in the *WebSecurityConfiguration* class.

Disable CSRF

```
...
@Bean
public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
    return http
        .csrf().disable() ①
        .authorizeExchange()
        .matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
    ...
```

① Add this line to disable CSRF.

Restart the application and retry to borrow a book. This time the request should be successful.

> Do not disable CSRF on productive servers if you use session cookies, otherwise you are vulnerable to CSRF attacks. You may safely disable CSRF for servers that use a stateless authentication approach with bearer tokens like for OAuth2 or OpenID Connect.

In this workshop step we added the authorization to web and method layers. So now for particular RESTful endpoints access is only permitted to users with special roles.

But how do you know that you have implemented all the authorization rules and did not leave a big security leak for your RESTful API? Or you may change some authorizations later by accident?

To be on a safer side here you need automatic testing. Yes, this can also be done for security! We will see how this works in the next workshop part.

5.4. Step 4: Security Testing

Now it is time to prove that we have implemented these authorization rules correctly with automatic testing.

We start testing the rules on method layer for all operations regarding books.

The tests will be implemented using the new JUnit 5 version as Spring 5.0 now supports this as well. In *BookServiceTest* class we also use the new convenience annotation *@SpringJUnitConfig* which is a shortcut of *@ExtendWith(value=SpringExtension.class)* and *@ContextConfiguration*.

As you can see in the following code only a small part is shown as a sample here to test the *BookService.create()* operation. Authorization should always be tested for positive **AND** negative test cases. Otherwise you probably miss an authorization constraint. Depending on the time left in the workshop you can add some more test cases as you like or just look into the completed application *04-library-server*.

BookServiceTest class

```
package com.example.library.server.business;

...

@DisplayName("Verify that book service")
@SpringJUnitConfig(LibraryServerApplication.class) ①
class BookServiceTest {

  @Autowired
  private BookService bookService;

  @MockBean ②
  private BookRepository bookRepository;

  @MockBean
  private UserRepository userRepository;

  @MockBean
  private IdGenerator idGenerator;

  @DisplayName("grants access to create a book for role 'CURATOR'")
  @Test
  @WithMockUser(roles = "CURATOR")
  void verifyCreateAccessIsGrantedForCurator() { ③
```

```
    when(bookRepository.insert(Mockito.<Mono<Book>>any())).thenReturn(Flux.just(new
Book()));
    StepVerifier.create(bookService.create(Mono.just(new
BookResource(UUID.randomUUID(),
                    "123456789", "title", "description",
Collections.singletonList("author"),
                    false, null)
        ))).verifyComplete();
  }

  @DisplayName("denies access to create a book for roles 'USER' and 'ADMIN'")
  @Test
  @WithMockUser(roles = {"USER", "ADMIN"})
  void verifyCreateAccessIsDeniedForUserAndAdmin() { ④
    StepVerifier.create(bookService.create(Mono.just(new
BookResource(UUID.randomUUID(),
            "123456789", "title", "description", Collections.singletonList("author"),
            false, null)
    ))).verifyError(AccessDeniedException.class);
  }

  @DisplayName("denies access to create a book for anonymous user")
  @Test
  void verifyCreateAccessIsDeniedForUnauthenticated() { ⑤
    StepVerifier.create(bookService.create(Mono.just(new
BookResource(UUID.randomUUID(),
            "123456789", "title", "description", Collections.singletonList("author"),
            false, null)
    ))).verifyError(AccessDeniedException.class);
  }

  ...

}
```

① As this is a JUnit 5 based integration test we use *@SpringJUnitConfig* to add spring JUnit 5 extension and configure the application context

② All data access (the repositories) is mocked

③ Positive test case of access control for creating books with role 'CURATOR'

④ Negative test case of access control for creating books with roles 'USER' or 'ADMIN'

⑤ Negative test case of access control for creating books with anonymous user

For sure you have to add similar tests as well for the user part.

UserServiceTest class

```
package com.example.library.server.business;

...
```

```
@DisplayName("Verify that user service")
@SpringJUnitConfig(LibraryServerApplication.class) ①
class UserServiceTest {

  @Autowired
  private UserService userService;

  @MockBean
  private UserRepository userRepository;

  @SuppressWarnings("unused")
  @MockBean
  private IdGenerator idGenerator;

  @DisplayName("grants access to find one user by email for anonymous user")
  @Test
  void verifyFindOneByEmailAccessIsGrantedForUnauthenticated() { ②
    when(userRepository.findOneByEmail(any())).thenReturn(Mono.just(new
User(UUID.randomUUID(),
            "test@example.com", "secret", "Max", "Maier",
            Collections.singletonList(Role.USER))));

StepVerifier.create(userService.findOneByEmail("test@example.com")).expectNextCount(1)
.verifyComplete();
  }

  @DisplayName("grants access to find one user by email for roles 'USER', 'CURATOR'
and 'ADMIN'")
  @Test
  @WithMockUser(roles = {"USER", "CURATOR", "ADMIN"})
  void verifyFindOneByEmailAccessIsGrantedForAllRoles() { ③
    when(userRepository.findOneByEmail(any())).thenReturn(Mono.just(new
User(UUID.randomUUID(),
            "test@example.com", "secret", "Max", "Maier",
            Collections.singletonList(Role.USER))));

StepVerifier.create(userService.findOneByEmail("test@example.com")).expectNextCount(1)
.verifyComplete();
  }

  @DisplayName("denies access to create a user for roles 'USER' and 'CURATOR'")
  @Test
  @WithMockUser(roles = {"USER", "CURATOR"})
  void verifyCreateAccessIsDeniedForUserAndCurator() { ④
    StepVerifier.create(userService.create(Mono.just(new
UserResource(UUID.randomUUID(),
            "test@example.com", "secret", "Max", "Maier",

Collections.singletonList(Role.USER))))).verifyError(AccessDeniedException.class);
  }
```

```
        ...
    }
```

① As this is a JUnit 5 based integration test we use *@SpringJUnitConfig* to add spring JUnit 5 extension and configure the application context

② Positive test case of access control for finding a user by email for anonymous user

③ Positive test case of access control for finding a user by email with all possible roles

④ Negative test case of access control for creating user with roles 'USER' or 'CURATOR'

The testing part is the last part of adding security to the reactive style of the *library-server* project.

> ℹ️ You find the completed code in project *04-library-server*.

In the last workshop part we will look at the new OAuth2 login client introduced in Spring Security 5.0. Unfortunately Spring does not support OAuth2 for reactive applications so far. That's why we will continue now with a "traditional" servlet based blocking alternative of the library-server.

5.5. Step 5: OAuth2/OpenID Connect

5.5.1. Introduction

OAuth2 is the base protocol for authorizing 3rd party authentication services for using business services in the internet like stackoverflow.com.
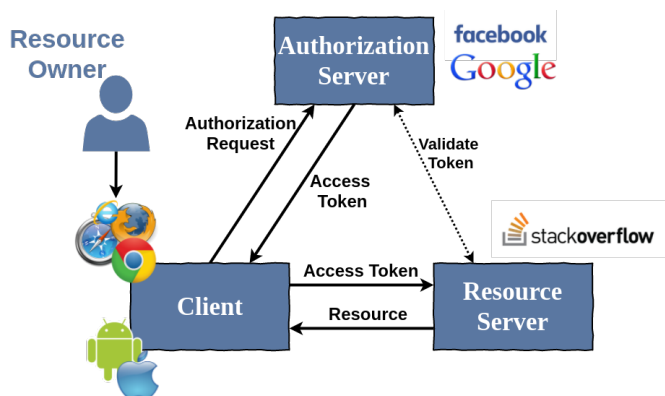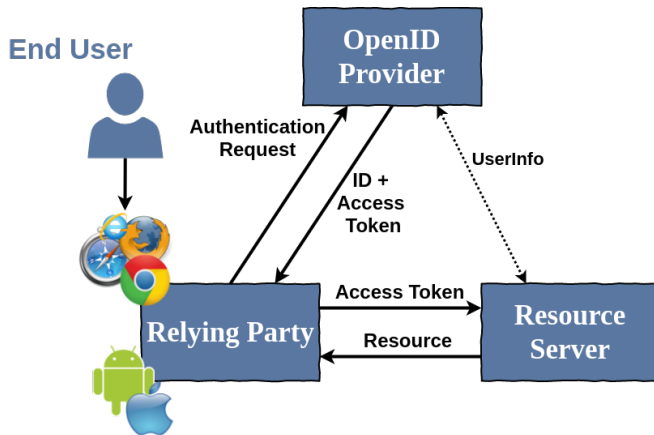

Authorizations are permitted via scopes that the user has to confirm before using the requested service.

Depending on the application type OAuth2 provides the following grants (flows):

- Authorization Code Grant
- Implicit Grant
- Client Credentials Grant
- Resource Owner Password Credentials Grant

OpenID Connect (OIDC) is buiuld upon OAuth2 and provides additional identity information to the scopes of OAuth2. For common enterprise applications OpenID Connect should be used. Nowadays

OAuth2 is frequently used together with providing identity in JSON web tokens (JWT). But only OIDC specifies the use of JWT, OAuth2 does not.

End User

OpenID Provider

Authentication Request

UserInfo

ID + Access Token

Access Token

Resource

Relying Party

Resource Server

OIDC adds an additional id token to the access token of OAuth2 and specifies a user info endpoint to retrieve further user information. With hybrid flow OIDC also adds one more grant (flow) to the existing ones of OAuth2.

5.5.2. OAuth2/OIDC in Spring Security 5

Spring Security 5.0 introduced new support for OAuth2/OpenID Connect (OIDC) directly in spring security.

In short Spring Security 5.0 adds a completely rewritten implementation for OAuth2/OIDC which now is largely based on a third party library Nimbus OAuth 2.0 SDK instead of implementing all these functionality directly in Spring itself.

Spring Security 5.0 only provides the client side for servlet-based clients.

Spring Security 5.1 adds the resource server support and reactive support for reactive clients and resource server as well.

Spring Security 5.2 will add a basic OAuth2/OIDC authorization server again (for local dev and demos but not for productive use).

Before Spring Security 5.0 and Spring Boot 2.0 to implement OAuth2 you needed the separate project module Spring Security OAuth2.

Now things have changed much, so it heavily depends now on the combination of Spring Security and Spring Boot versions that are used how to implement OAuth2/OIDC.

Therefore you have to be aware of different approaches for Spring Security 4.x/Spring Boot 1.5.x and Spring Security 5.x/Spring Boot 2.x.

Table 3. OAuth2 support in Spring Security + Spring Boot

Spring Security	Spring Boot	Client	Resource server	Authorization server	Reactive (WebFlux)
4.x	1.5.x	X[1]	X[1]	X[1]	—
5.0	2.0.x	X[2]	(X)[3]	(X)[3]	—
5.1	2.1.x	X[2]	X[4]	(X)[3]	X[5]

| 5.2 | 2.x | X[2] | X[4] | X[6] | X[5] |

[1] Spring Boot auto-config and separate Spring Security OAuth project
[2] New rewritten OAuth2 login client included in Spring Security 5.0
[3] No direct support in Spring Security 5.0/Spring Boot 2.0. For auto-configuration with Spring Boot 2.0 you still have to use the separate Spring Security OAuth project together with Spring Security OAuth2 Boot compatibilty project
[4] New refactored support for resource server as part of Spring Security 5.1
[5] OAuth2 login client and resource server with reactive support as part of Spring Security 5.1.
[6] New OAuth2 authorization server is planned as part of Spring Security 5.2

> 💡 The OAuth2/OpenID Connect Authorization Server provided by Spring Security 5.2 will mainly suit for fast prototyping and demo purposes. For production please use one of the officially certified products like for example KeyCloak, UAA or Okta.

You can find more information on building OAuth2 secured microservices with Spring Boot **1.5.x** in

- Spring Boot 1.5 Reference Documentation
- Spring Security OAuth2 Developers Guide

You can find more information on building OAuth2 secured microservices with Spring Boot **2.0** in

- Spring Boot 2.1 Reference Documentation
- Spring Security OAuth2/OIDC Login Client Reference Documentation
- Spring Security OAuth2/OIDC Resource Server Reference Documentation
- Spring Security OAuth Boot 2 Auto-config Documentation
- Spring Security OAuth2 Developers Guide

In this workshop we will now look at what Spring Security 5.1 provides as new OAuth2/OIDC Login Client and Resource Server - In a reactive way.

5.5.3. What we will build

In the *05-oauth2* project you will be provided the following sub-projects:

- **authorizationserver**: This is the OAuth2 authorization server which holds all users with their credentials. This is still based on Spring Boot 1.5.x.
- **initial-library-server**: The reactive version of the library server (same as previous workshop step *04-library-server*)
- **initial-oauth2-login-client**: Initial code for this workshop part to implement the new OAuth2 Login Client
- **oauth2-login-client**: Complete code of the new OAuth2 Login Client (for reference)
- **oauth2-library-server**: Complete code of the reactive library OAuth2 resource server

OAuth2
Authorization Server

User
Microservice

UI
Microservice

OAuth2
Login Client

Library
Microservice

OAuth2
Resource Server
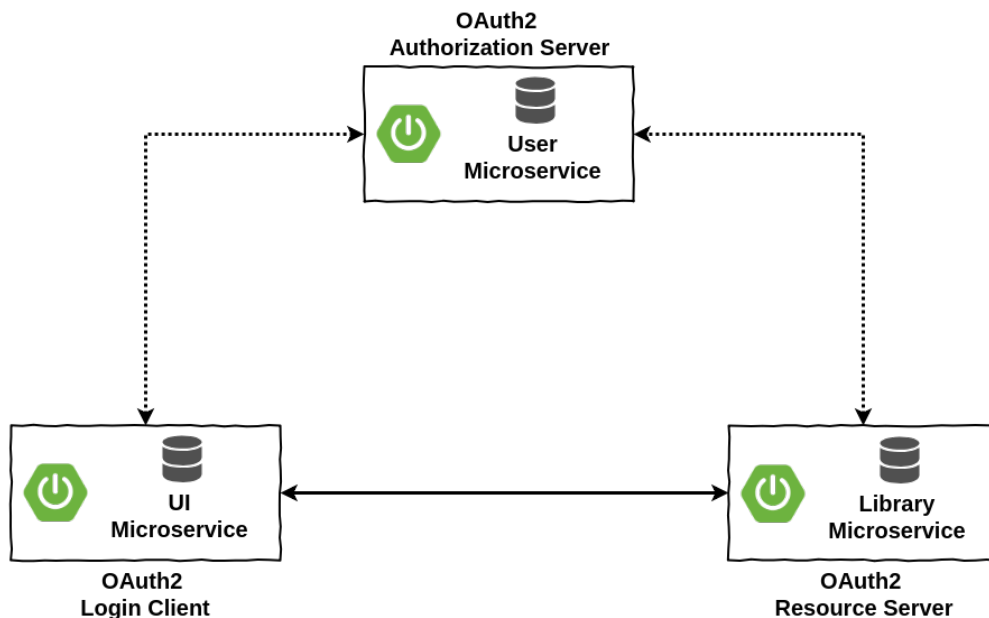
These microservices have to be configured to be reachable via the following URL addresses (Port 8080 is the default port in spring boot).

Table 4. Microservice URL Adresses

Microservice	URL
Authorization Server	http://localhost:9999/users
Login Client	http://localhost:8081
Library-Server (Resource Server)	http://localhost:8080

So now let's start. Again, you will just use the provided *AuthorizationServer* and *Library-Server* and start to implement a OAuth2 login client based on the *initial-oauth2-login-client* project.

First some information about how to start the required other components *AuthorizationServer* and *Library-Server*.

5.5.4. Authorization Server

For this workshop a simple authorization server issuing JWT OAuth2 tokens is provided. As there is no support for an authorization server in Spring Boot 2.0 this component has been implemented using Spring Boot 1.5.x.

> You may look into the spring boot reference documentation Spring Boot 1.5 Reference Documentation on how to implement an authorization server.

To ensure OAuth2 authorization code grant works correctly with the other components the end points of the authorization server must be as follows:

Table 5. Authorization Server Endpoints

Endpoint	Description	Caller
/oauth/authorize	Authorization endpoint (for login and client authorization)	Client

/oauth/token	Token endpoint (exchanges given authorization code for access token)	Client
/oauth/check_token	Check token endpoint (returns internal contents for access token)	Resource Server

> ❗ To prevent conflicts with different JSESSION cookies the authorization server must run on a separate context path (not '/'). In our example please use '/users' as context path. In spring boot this can be achieved by the *server.context* property

To start the authorization server simply run the class *AuthorizationServerApplication* in project *05-oauth2-client/authorizationserver*.

5.5.5. Resource Server (Library-Server)

For this workshop part the well-known library-server application has been changed again into a blocking servlet-based web application that also acts now as a OAuth2 resource server. This was required as there is no support yet in Spring for a reactive resource server.

As the library-server is already build using Spring Boot 2.0 the compatibility module spring security oauth2 boot was used for auto configuration using Spring Boot 2.0 with the old Spring Security OAuth project.

> 💡 You may look into the spring security oauth2 boot reference documentation Spring OAuth2 Autoconfig Reference Documentation on how to implement a resource server.

To start the resource server simply run the class *LibraryServerApplication* in project *05-oauth2-client/library-server*.

In the following paragraphs we now proceed to the workshop part where your interaction is required again: The OAuth2 login client.

5.5.6. OAuth2 Login Client

Gradle dependencies

To use the new OAuth2 client support of Spring Security 5.0 you have to add the following required dependencies to the existing gradle build file.

gradle.build file

```
dependencies {
    ...
    compile('org.springframework.security:spring-security-oauth2-client') ①
    compile('org.springframework.security:spring-security-oauth2-jose') ②
    ...
}
```

① This contains core classes and interfaces that provide support for the OAuth 2.0 Authorization Framework and for OpenID Connect Core 1.0

② This contains Spring Security's support for the JOSE (Javascript Object Signing and Encryption) framework. This is needed to support for example JSON Web Token (JWT)

> ℹ These dependencies already have been added to the initial project.

Implementation steps

First step is to configure an OAuth2 login client. For this you have to register the corresponding identity server/authorization server to use. Here you have two possibilities:

1. You can just use one of the predefined ones (Facebook, Google, etc.)

2. You register your own custom server

Spring security provides the enumeration *CommonOAuth2Provider* which defines registration details for a lot of well known identity providers.

CommonOAuth2Provider class

```
package org.springframework.security.config.oauth2.client;
...
public enum CommonOAuth2Provider {

    GOOGLE {

        @Override
        public Builder getBuilder(String registrationId) {
            ClientRegistration.Builder builder = getBuilder(registrationId,
                    ClientAuthenticationMethod.BASIC, DEFAULT_LOGIN_REDIRECT_URL);
            builder.scope("openid", "profile", "email");
            builder.authorizationUri("https://accounts.google.com/o/oauth2/v2/auth");
            builder.tokenUri("https://www.googleapis.com/oauth2/v4/token");
            builder.jwkSetUri("https://www.googleapis.com/oauth2/v3/certs");
            builder.userInfoUri("https://www.googleapis.com/oauth2/v3/userinfo");
            builder.userNameAttributeName(IdTokenClaimNames.SUB);
            builder.clientName("Google");
            return builder;
        }
    },

    GITHUB {

        @Override
        public Builder getBuilder(String registrationId) {
            ...
        }
    },

    FACEBOOK {

        @Override
        public Builder getBuilder(String registrationId) {
            ...
        }
    },
    ...
}
```

To use one of these providers is quite easy. Just reference the enumeration constant as the provider.

Google provider properties class

```
spring:
  security:
    oauth2:
      client:
        registration:
          google-login: ①
            provider: google ②
            client-id: google-client-id
            client-secret: google-client-secret
```

① The registration id is set to *google-login*

② The provider is set to the predefined *google* client which points to *CommonOAuth2Provider.GOOGLE*

But in this workshop we will focus on the second possibility and use our own local authorization server.
To achieve this we add the following sections to the *application.yml* file.

application.yml client configuration

```
spring:
  security:
    oauth2:
      client:
        registration:
          login-client:
            provider: local-authserver ①
            client-id: library-client ②
            client-secret: secret ③
            client-authentication-method: basic ④
            authorization-grant-type: authorization_code ⑤
            scope: profile ⑥
            client-name: Login Client ⑦
            redirect-uri-template: "{baseUrl}/login/oauth2/code/{registrationId}" ⑧
        provider:
          local-authserver: ⑨
            authorization-uri: http://localhost:9999/users/oauth/authorize ⑩
            token-uri: http://localhost:9999/users/oauth/token ⑪
            user-info-uri: http://localhost:9999/users/resources/userinfo ⑫
            user-name-attribute: id ⑬
```

① References the registered identity provider/authorization server to use for this OAuth2 client

② The client id to use at the authorization server

③ The client secret to use at the authorization server

④ The authentication mode to use at the authorization server (may be basic or post)

⑤ The required OAuth2 flow, here the most commonly used flow *authorization_code* is used

⑥ The required OAuth2 scope to use at the authorization server

⑦ A human readable name for this OAuth2 login client configuration

⑧ The pattern to build the required redirect URL back from authorization server to this client

⑨ Name of the identity provider (in this case the local OAuth2 authorization server)

⑩ The URI for authorizing the user at the authorization server (getting the authorization code)

⑪ The URI for getting a JWT token from the authorization server (exchanging the authorization code)

⑫ The URI to access the claims/attributes of the authenticated end-user using the JWT access token from the authorization server

⑬ The name of the attribute returned in the user info that references the name or identifier of the end-user. (retrieved from the user-info-uri endpoint)

As the library-server is now configured as an OAuth2 resource server it requires a valid JWT token to successfully call the */books* endpoint now.

From client side we use the *RestTemplate* for calls to the RESTful service. To support JWT tokens in calls we have to add a client interceptor to the *RestTemplate*. The following code snippet shows how this is done:

RestTemplateConfiguration class

```
package com.example.oauth2loginclient.config;

...

@Configuration
public class RestTemplateConfiguration {

  @Bean
  RestTemplate restTemplate(OAuth2AuthorizedClientService clientService) {
    return new RestTemplateBuilder()
        .interceptors( ①
            (ClientHttpRequestInterceptor)
                (httpRequest, bytes, execution) -> {
                  Authentication authentication =
                      SecurityContextHolder.getContext().getAuthentication(); ②
                  if (authentication instanceof OAuth2AuthenticationToken) {
                    OAuth2AuthenticationToken token =
                            (OAuth2AuthenticationToken) authentication;

                    OAuth2AuthorizedClient client =
                        clientService.loadAuthorizedClient(
                            token.getAuthorizedClientRegistrationId(),
token.getName());

                    httpRequest
                        .getHeaders()
                        .add(
                            HttpHeaders.AUTHORIZATION, ③
                            String.format("Bearer %s",
client.getAccessToken().getTokenValue()));
                  }
                  return execution.execute(httpRequest, bytes);
                })
        .build();
  }
}
```

① Adds an client side interceptor to the rest template for adding the OAuth2 access token as header

② Gets the authentication object with OAuth2 credentials from the security context

③ Adds the OAuth2 access token (base64 encoded JWT) as *Authorization* header to all requests

Run all the components

Finally start all three components:

- Run *AuthorizationServerApplication* class in project *05-oauth2-client/authorizationserver*

- Run *LibraryServerApplication* class in project *05-oauth2-client/library-server*

- Run *OAuth2LoginClientApplication* class in project *05-oauth2-client/initial-oauth2-login-client*

Now when you access localhost:8081 you should be redirected to the authorization server. After logging in you should get the current authenticated user infos back from authorization server.

Here you can log in using one of these predefined users:

Table 6. User credentials

User	Password	Roles
user@example.com	user_4workshop	USER
curator@example.com	curator_4workshop	USER, CURATOR
admin@example.com	admin_4workshop	USER, CURATOR, ADMIN

You can now access localhost:8081/books as well. This returns the book list from the library-server (resource server).

> You find the completed code in project *05-oauth2-client/oauth2-login-client*.

This concludes our Spring Security 5.1 hands-on workshop. I hope you learned a lot regarding security and especially Spring Security 5.1.

> If you have feedback for this workshop, suggestions for improvements or you want me to conduct this workshop somewhere else please do not hesitate to contact me via
>
> - Mail: andreas.falk@novatec-gmbh.de
> - Twitter: @andifalk
> - LinkedIn: andifalk

Thank YOU very much for being part of this workshop :-)

Appendix A: Online References

- OWASP Top 10 2017
- OWASP ProActive Controls 2018
- OWASP Testing Guide
- OAuth2 Specifications
- OpenID Connect 1.0 Specification
- Spring Boot 1.5 Reference Guide
- Spring Boot 2.0 Reference Guide
- Spring Security 4.x Reference Guide
- Spring Security 5.x Reference Guide
- *Legacy* Spring Security OAuth Reference Guide
- *Legacy* Spring Security OAuth2 Boot Reference Guide
- Spring Security OAuth2 Client for WebFlux Sample
- Reactive Spring Security 5 Workshop Code

Appendix B: Book References

- Iron-Clad Java: Building Secure Web Applications (Oracle Press, ISBN: 978-0071835886)

- OAuth 2 in Action (Manning Publications, ISBN: 978-1617293276)

- Spring in Action 5th Edition (Manning Publications, ISBN: 978-1617294945