

Reactive Spring Security 5.1 Hands-On Workshop

Andreas Falk

Table of Contents

1. Introduction	2
1.1. Requirements for this workshop	2
2. Common Web Security Risks	4
3. The workshop application	5
3.1. Reactive Systems & Streams	5
3.1.1. Project Reactor	6
3.1.2. Spring WebFlux	7
4. Workshop Organization	8
5. Workshop Steps	9
5.1. Step 1: Auto Configuration	9
5.1.1. Login	9
5.1.2. Common Security Problems	10
5.1.3. Logout	11
5.2. Step 2: Customize Authentication	11
5.2.1. WebFilter	11
5.2.2. WebFilterChainProxy	12
5.2.3. Encrypting Passwords	15
5.2.4. Persistent User Storage	18
5.2.5. Automatic Password Encryption Updates	21
5.3. Step 3: Add Authorization	21
5.4. Step 4: Security Testing	30
5.5. Step 5: OAuth2/OpenID Connect	33
5.5.1. OAuth 2.0	33
5.5.2. OpenID Connect 1.0	34
5.5.3. Tokens in OIDC and OAuth 2.0	34
5.5.4. OAuth2/OIDC in Spring Security 5	34
5.5.5. What we will build	36
5.5.6. UAA Identity Server	37
5.5.7. Resource Server (Library-Server)	38
5.5.8. OAuth2 Login Client	39
6. Feedback	46
Appendix A: Online References	47
Appendix B: Book References	48



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 JDK (Java 8, 9 or 11 are supported and tested)
- 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]	U	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]	U	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.]

The [Open Web Application Security Project](#) has plenty of further free resources available.

As a developer you may also have a look into the [OWASP ProActive Controls](#) document which describes how to develop your applications using good security patterns. To help getting all security requirements right for your project and how to test these the [Application Security Verification Standard](#) can help you here.



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 Mongoddb 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.

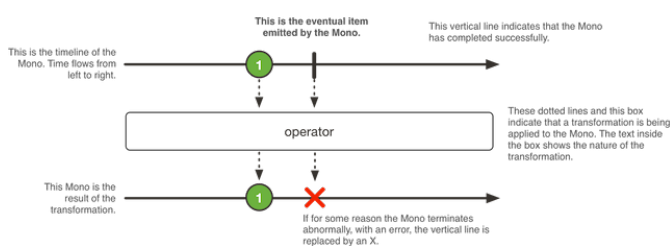


Figure 1. Mono, an Asynchronous 0-1 Result (source: projectreactor.io)

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.

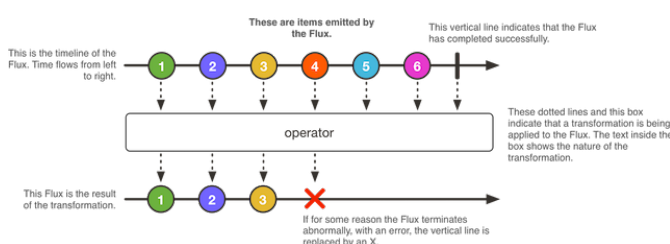


Figure 2. Flux, an Asynchronous Sequence of 0-N Items (source: projectreactor.io)

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 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:

- **reactive-playground:** This project provides a playground to try reactive streams with project reactor
- **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/OIDC login client and resource server for interacting with an identity server for login

Chapter 5. Workshop Steps

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



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*.

5.1.1. Login



A screenshot of a web browser showing a login form. The form has a title "Please sign in". Below the title are two input fields: the first is labeled "user" and contains the text "user"; the second is a password field filled with dots. To the right of each input field is a small icon of a speech bubble with a question mark. Below the input fields is a blue button with the text "Sign in".

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.

5.1.2. Common Security Problems

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

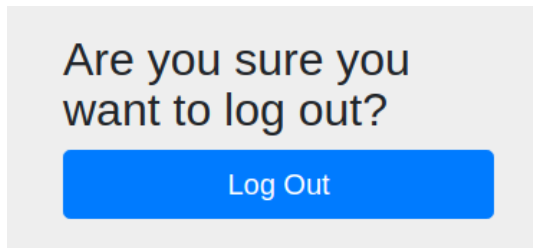
- [Session Fixation](#): Session Fixation is an attack that permits an attacker to hijack a valid user session.
If you want to learn more about this please read the [Session Fixation page at OWASP](#)
- [Cross Site Request Forgery \(CSRF\)](#): Cross-Site Request Forgery (CSRF) is an attack that forces 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  
Referrer-Policy: no-referrer  
X-Content-Type-Options: nosniff  
X-Frame-Options: DENY  
X-XSS-Protection: 1 ; mode=block
```

5.1.3. Logout

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.



This concludes the first step.



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.

Before we start let's look into some internal details how spring security works for the reactive web stack.

5.2.1. WebFilter

Like the `javax.servlet.Filter` in the blocking servlet-based web stack there is a comparable functionality in the reactive world: The *WebFilter*.

WebFilter

```
public interface WebFilter {

    /**
     * Process the Web request and (optionally) delegate to the next
     * {@code WebFilter} through the given {@link WebFilterChain}.
     * @param exchange the current server exchange
     * @param chain provides a way to delegate to the next filter
     * @return {@code Mono<Void>} to indicate when request processing is complete
     */
    Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain);

}
```

By using the *WebFilter* you can add functionality that called around each request and response.

Table 1. Spring Security WebFilter

Filter	Description
AuthenticationWebFilter	Performs authentication of a particular request
AuthorizationWebFilter	Determines if an authenticated user has access to a specific object
CorsWebFilter	Handles CORS preflight requests and intercepts
CsrfWebFilter	Applies CSRF protection using a synchronizer token pattern.

To see how such a *WebFilter* works we will implement a simple *LoggingWebFilter*:

LoggingWebFilter

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

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import org.springframework.web.server.WebFilter;
import org.springframework.web.server.WebFilterChain;
import reactor.core.publisher.Mono;

@Component
public class LoggingWebFilter implements WebFilter {

    private static Logger LOGGER = LoggerFactory.getLogger(LoggingWebFilter.class);

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {
        LOGGER.info("Request {} called", exchange.getRequest().getPath().value());
        return chain.filter(exchange);
    }
}
```

5.2.2. WebFilterChainProxy

In step 1 we just used the auto configuration of Spring Boot. This configured default security settings as follows:

```
@Configuration
class WebFluxSecurityConfiguration {
    ...

    /**
     * The default {@link ServerHttpSecurity} configuration.
     * @param http
     * @return
     */
    private SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http)
    {
        http
            .authorizeExchange()
                .anyExchange().authenticated();

        if (isOAuth2Present && OAuth2ClasspathGuard.shouldConfigure(this.context)) {
            OAuth2ClasspathGuard.configure(this.context, http);
        } else {
            http
                .httpBasic().and()
                .formLogin();
        }

        SecurityWebFilterChain result = http.build();
        return result;
    }

    ...
}
```

As you can see this uses a *SecurityWebFilterChain* as central component.

```
/**
 * Defines a filter chain which is capable of being matched against a {@link
ServerWebExchange} in order to decide
 * whether it applies to that request.
 *
 * @author Rob Winch
 * @since 5.0
 */
public interface SecurityWebFilterChain {

    /**
     * Determines if this {@link SecurityWebFilterChain} matches the provided {@link
ServerWebExchange}
     * @param exchange the {@link ServerWebExchange}
     * @return true if it matches, else false
     */
    Mono<Boolean> matches(ServerWebExchange exchange);

    /**
     * The {@link WebFilter} to use
     * @return
     */
    Flux<WebFilter> getWebFilters();
}
```

To customize the spring security configuration you have to implement one or more of *SecurityWebFilterChain* configuration methods.

These are handled centrally by the *WebFilterChainProxy* class.


```
public class WebFilterChainProxy implements WebFilter {
    private final List<SecurityWebFilterChain> filters;

    public WebFilterChainProxy(List<SecurityWebFilterChain> filters) {
        this.filters = filters;
    }

    public WebFilterChainProxy(SecurityWebFilterChain... filters) {
        this.filters = Arrays.asList(filters);
    }

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) { ❶
        return Flux.fromIterable(this.filters)
            .filterWhen( securityWebFilterChain ->
securityWebFilterChain.matches(exchange))
            .next()
            .switchIfEmpty(chain.filter(exchange).then(Mono.empty()))
            .flatMap( securityWebFilterChain ->
securityWebFilterChain.getWebFilters()
                .collectList()
            )
            .map( filters -> new FilteringWebHandler(webHandler ->
chain.filter(webHandler), filters))
            .map( handler -> new DefaultWebFilterChain(handler) )
            .flatMap( securedChain -> securedChain.filter(exchange));
    }
}
```

❶ Central point for spring security to step into reactive web requests

5.2.3. Encrypting Passwords

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*.

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

public class PasswordEncoderFactories {
    ...
    public static PasswordEncoder createDelegatingPasswordEncoder() {
        String encodingId = "bcrypt"; ❶
        Map<String, PasswordEncoder> encoders = new HashMap<>();
        encoders.put(encodingId, new BCryptPasswordEncoder()); ❷
        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);
    }
    ...
}
```

❶ BCrypt is the default for encrypting passwords

❷ 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.

```
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)

5.2.4. Persistent User Storage

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

```
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;
    }

    @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 2. User credentials

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

5.2.5. Automatic Password Encryption Updates

TODO

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 3. Authorization rules for library-server

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


```

package com.example.library.server.config;

...

@EnableWebFluxSecurity
@EnableReactiveMethodSecurity ❶
public class WebSecurityConfiguration {

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

            .matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll() ❸
                .matchers(EndpointRequest.to("health")).permitAll() ❹
                .matchers(EndpointRequest.to("info")).permitAll()
                .matchers(EndpointRequest.toAnyEndpoint()).hasRole(Role.ADMIN.name())
❺

                .pathMatchers(HttpMethod.POST, "/books").hasRole(Role.CURATOR.name())
❻

                .pathMatchers(HttpMethod.DELETE,
"/books").hasRole(Role.CURATOR.name())
                .pathMatchers("/users/**").hasRole(Role.ADMIN.name()) ❼
                .anyExchange().authenticated() ❽
                .and()
                .httpBasic().and().formLogin().and() ❾
                .logout().logoutSuccessHandler(logoutSuccessHandler()) ❿
                .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();
    }
}

```

❶ This adds support for method level authorization

❷ 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 *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).

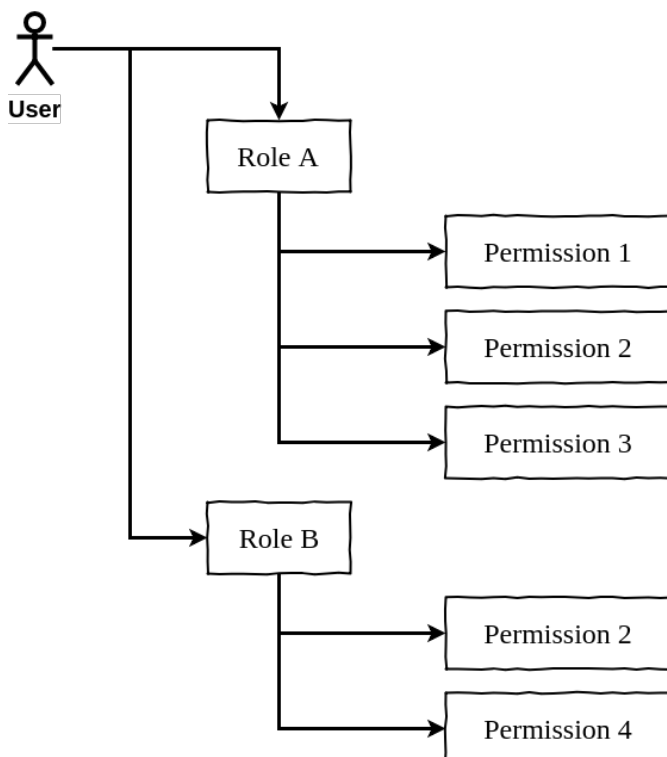


Figure 3. Roles and Permissions

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.

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

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

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

    ...

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

    ...

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

    ...
}
```

- ❶ In general all users (having either of these 3 roles) can access RESTful services for books
- ❷ Only users having role 'CURATOR' can access this RESTful service to create books
- ❸ 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.

```
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.

```
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) { ❶
        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) { ❷
        return bookService.returnById(bookId, user.getUserResource().getId());
    }

    ...
}
```

- ❶ Now that we have an authenticated user context we can add the current user as the one to borrow a book
- ❷ 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.



You find the completed code in project *03-library-server*.

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) ❶
class BookServiceTest {

    @Autowired
    private BookService bookService;

    @MockBean ❷
    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() { ❸
```



```

        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) ❶
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() { ❷
        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() { ❸
        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() { ❹
        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.



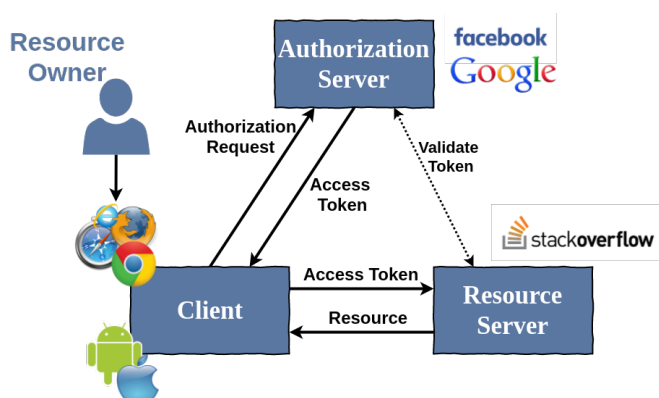
You find the completed code in project *04-library-server*.

In the last workshop part we will look at the new OAuth2 login client and resource server introduced in Spring Security 5.0 and 5.1.

5.5. Step 5: OAuth2/OpenID Connect

5.5.1. OAuth 2.0

OAuth 2.0 is the base protocol for authorizing 3rd party authentication services for using business services in the internet like [stackoverflow](#).



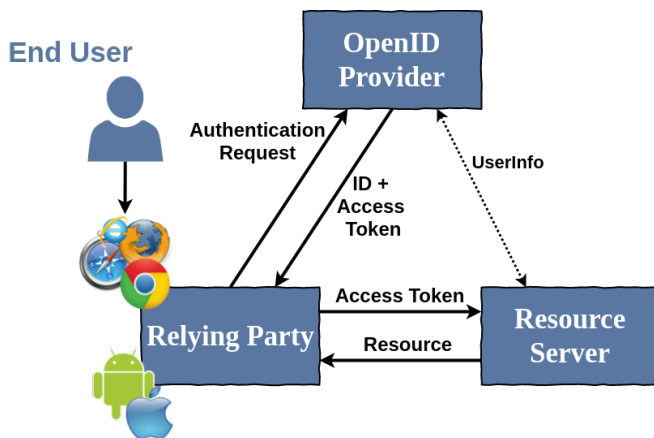
Authorizations are permitted via scopes that the user has to confirm before using the requested service.

Depending on the application type OAuth 2.0 provides the following grants (flows):

- [Authorization Code Grant](#)
- [Authorization Code Grant with PKCE](#)
- [Implicit Grant](#)
- [Resource Owner Password Credentials Grant](#)
- [Client Credentials Grant](#)

5.5.2. OpenID Connect 1.0

[OpenID Connect 1.0 \(OIDC\)](#) is build upon [OAuth2](#) and provides additional identity information to OAuth2. For common enterprise applications that typically require authentication OpenID Connect should be used. [OIDC](#) adds [JSON web tokens \(JWT\)](#) as mandatory format for id tokens to the spec. In OAuth2 the format of [bearer tokens](#) is not specified.



OIDC adds an id token in addition to the access token of OAuth2 and specifies a user info endpoint to retrieve further user information using the access token.

OIDC supports the following grant flows:

- [Authorization Code Flow](#)
- [Implicit Flow](#)
- [Hybrid Flow](#)

5.5.3. Tokens in OIDC and OAuth 2.0

Tokens can be used in two ways:

1. Self-contained token (containing all information directly in token payload, e.g. JWT)
2. Reference token (token is used to look up further information)

5.5.4. 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 adds client support for authorization code flow with PKCE.

Spring Security 5.3 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 4. 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	2.1.2	X ²	X ⁴	(X) ³	X ⁵
5.2	2.2.0	X ²	X ⁴	(X) ³	X ⁵
5.3	2.3.0	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 compatibility 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 5.3 will mainly suit for fast prototyping and demo purposes. For production please use one of the [officially certified](#) products like for example [KeyCloak](#), [UAA](#), [IdentityServer](#), [Auth0](#) 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 **2.1** and Spring Security **5.1** 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.

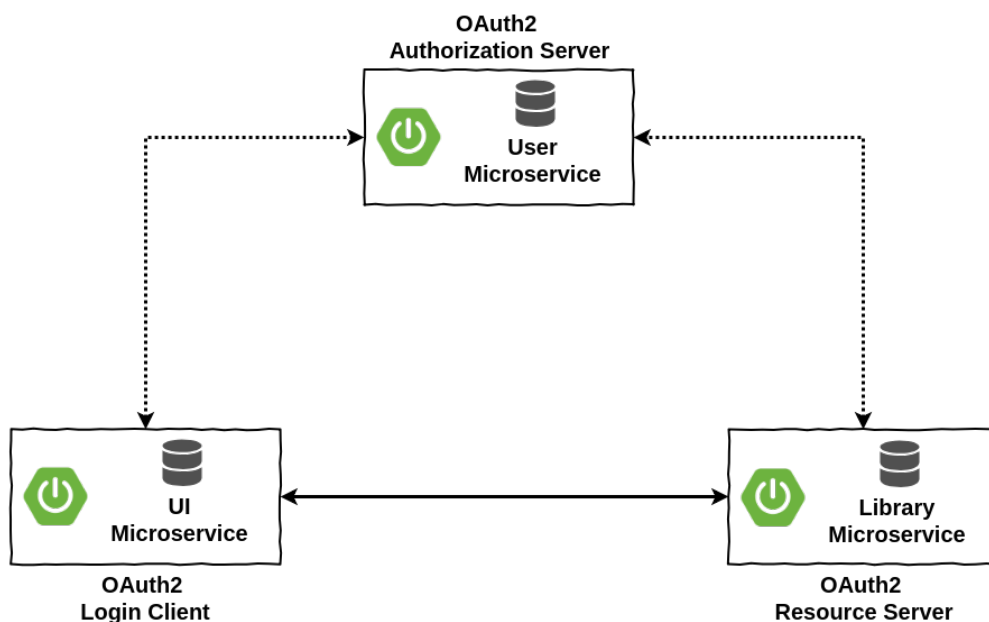
5.5.5. What we will build

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

- **initial-library-server**: The reactive version of the library server (similar to 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



The spring implementation of the authorization server previously used (based on Spring Boot 1.5.x) is not fully compliant with OIDC and therefore not usable any more with OAuth2/OIDC implementation of Spring Security 5.1.



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

Table 5. Microservice URL Adresses

Microservice	URL
Identity Management Server (UAA)	http://localhost:8090/uaa
Library Client (OAuth2 Login Client)	http://localhost:8081
Library-Server (OAuth2 Resource Server)	http://localhost:8080

So now let's start. Again, you will just use the provided *uaa* identity management server, the *initial-library-server* and the *initial-oauth2-login-client* as starting point and implement a OAuth2 resource

server and login client based on the project.

But first read important information about how to start the required *uaa* identity management server.

5.5.6. UAA Identity Server

For this workshop the OpenID Connect certified [UAA identity server](#) is provided. UAA is an abbreviation for CloudFoundry User Account and Authentication. This server is issuing OAuth2/OIDC JSON web tokens.



You may look into [OpenID Connect certified products](#) to find a suitable identity management server for your project.

To start the UAA component you need an installed [Apache Tomcat](#) server.

1. Please [download](#) most current version of the Apache Tomcat 8.x line and extract the downloaded archive in a directory of your choice.
2. Open the file `05-oauth2/uaa/build.gradle` and replace the path entry for `tomcatHomeDir` with the directory you have extracted the Apache Tomcat files into.
3. Execute `./gradlew downloadUAA` to download required war file for UAA
4. Execute `./gradlew cargoRunLocal` to start the UAA server

Wait until the UAA server has been started. If you see the output `Press Ctrl-C to stop the container` then UAA is ready. You can also look into the file `05-oauth2/uaa/uaa-server.log` to see the corresponding log output for UAA.

To see the configuration please open the following url in your web browser: (Every OpenID Connect compliant identity server must provide a page at the endpoint `.../.well-known/openid-configuration`

<http://localhost:8090/uaa/oauth/token/.well-known/openid-configuration>

The important information provided by this is:

Table 6. Identity Server Configuration

Entry	Description	Value
issuer	Issuer url for issued tokens by this identity server	http://localhost:8090/uaa/oauth/token
authorization_endpoint	Handles authorization, usually asking for credentials and returns an authorization code	http://localhost:8090/uaa/oauth/authorize
token_endpoint	Token endpoint (exchanges given authorization code for access token)	http://localhost:8090/uaa/oauth/token
userinfo_endpoint	Endpoint for requesting further user information	http://localhost:8090/uaa/userinfo

jwtks_uri	Uri for loading public keys to verify signatures of JSON web tokens	http://localhost:8090/uaa/token_keys
-----------	---	---

5.5.7. Resource Server (Library-Server)

For this workshop part the well-known library-server application is used and will be extended to act as a OAuth2 resource server.

Gradle dependencies

To use the new OAuth2 resource server support of Spring Security 5.1 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-resource-server') ①
    compile('org.springframework.security:spring-security-oauth2-jose') ②
    ...
}
```

- ① This contains all code to build an OAuth 2.0/OIDC resource server
- 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)



These dependencies already have been added to the initial project.



You may look into the spring security oauth2 boot reference documentation [Spring Boot 2.1 Reference Documentation](#) and the [Spring Security 5.1.1 Reference Documentation](#) on how to implement a resource server.

Implementation steps

First step is to configure an OAuth2 resource server. For this you have to register the corresponding identity server/authorization server to use.

application.yml file

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8090/uaa/oauth/token ①
```

- ① The issuer url is used to look up the well known configuration page to get all required configuration settings to set up a resource server


```
@EnableWebFluxSecurity
@EnableReactiveMethodSecurity
public class WebSecurityConfiguration {

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http
            .csrf().disable()
            .authorizeExchange()

            .matchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
            .matchers(EndpointRequest.to("health")).permitAll()
            .matchers(EndpointRequest.to("info")).permitAll()
            .matchers(EndpointRequest.toAnyEndpoint()).hasAuthority("SCOPE_admin")

            ① .pathMatchers(HttpMethod.POST, "/books").hasAuthority("SCOPE_curator")
            .pathMatchers(HttpMethod.DELETE,
"/books").hasAuthority("SCOPE_curator")
            .pathMatchers("/users/**").hasAuthority("SCOPE_admin")
            .pathMatchers("/books/**").hasAuthority("SCOPE_user")
            .anyExchange().authenticated()
            .and()
            .oauth2ResourceServer() ②
            .jwt(); ③
        return http.build();
    }
}
```

① OAuth2 scopes are automatically mapped to *SCOPE_xxx* authorities (like the standard *ROLE_xxx* ones)

② Autoconfiguration for an OAuth2 resource server

③ Configures JSON web token (JWT) handling for this resource server

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

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

5.5.8. 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.

```
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)



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.

```
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.

```
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:
          library-client: ①
          provider: okta ②
          client-id: 0oajbryix1d8ABX1i0h7 ③
          client-secret: IO_t-KoGka55dz0AVx94mra0ig4zclp4ri4Y92v1 ④
          redirect-uri-template: "{baseUrl}/login/oauth2/code/{registrationId}" ⑤
          scope: openid, profile, email, offline_access ⑥
        provider:
          okta: ⑦
          issuer-uri: https://dev-667216.oktapreview.com/oauth2/ausjbx4pq5Wg0kZx10h7
```

⑧

① The registration id for the client

② References the registered identity provider to use for this OAuth2 client

③ The client id to use at the identity provider

④ The client secret to use at the identity provider

⑤ The pattern to build the required redirect URL back from authorization server to this client

⑥ The required OAuth2 scopes to use at the identity provider

⑦ Name of the identity provider (Okta identity provider service)

⑧ The issuer URI for tokens (is used to read the identity provider configuration from `https://…​`

8203;/well-known/openid-configuration)

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 new *WebClient* for calls to the RESTful service. *WebClient* is the successor of *RestTemplate* and works for both worlds (Servlet-based and reactive).

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:

WebClientConfiguration class

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

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.oauth2.client.registration.ReactiveClientRegistrationRepository;
import org.springframework.security.oauth2.client.web.reactive.function.client.ServerOAuth2AuthorizedClientExchangeFilterFunction;
import org.springframework.security.oauth2.client.web.server.ServerOAuth2AuthorizedClientRepository;
import org.springframework.web.reactive.function.client.WebClient;

@Configuration
public class WebClientConfiguration {

    @Bean
    WebClient webClient(ReactiveClientRegistrationRepository
clientRegistrationRepository,
                        ServerOAuth2AuthorizedClientRepository
authorizedClientRepository) {
        ServerOAuth2AuthorizedClientExchangeFilterFunction oauth = ❶
            new
ServerOAuth2AuthorizedClientExchangeFilterFunction(clientRegistrationRepository,
authorizedClientRepository);
        oauth.setDefaultClientId("library-app"); ❷
        return WebClient.builder()
            .filter(oauth) ❸
            .build();
    }
}
```

- ❶ Creates a filter for handling all the OAuth2 token stuff (i.e. initiating the OAuth2 code grant flow)
- ❷ Registration id for client for automatic token handling

③ Add the filter to webclient

Finally we need an updated client side security configuration to allow client endpoints and enable the OAuth2 client features:

SecurityConfiguration class

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

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.reactive.EnableWebFluxSecurity;
import org.springframework.security.config.web.server.ServerHttpSecurity;
import org.springframework.security.web.server.SecurityWebFilterChain;

@EnableWebFluxSecurity
@Configuration
public class SecurityConfiguration {

    @Bean
    SecurityWebFilterChain configure(ServerHttpSecurity http) throws Exception {
        http.authorizeExchange() ①
            .anyExchange().authenticated() ②
            .and()
            .oauth2Login() ③
            .and()
            .oauth2Client(); ④
        return http.build();
    }
}
```

① Starting point for authorization rules (analogous to authorizeRequests in servlet world)

② Secure all client side endpoints

③ Add OAuth2/OIDC login feature

④ Add OAuth2/OIDC standard client feature

Run all the components

Finally start all three components:

- Run *UAA* server in project *05-oauth2/uaa*
- Run *LibraryServerApplication* class in project *05-oauth2/initial-library-server*
- Run *OAuth2LoginClientApplication* class in project *05-oauth2/initial-oauth2-login-client*

Now when you access localhost:8081/userinfo you should be redirected to the UAA identity server. After logging in you should get the current authenticated user info back from identity server.

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

Table 7. User credentials

User	Password	Roles
user@example.com	Library_access#1	USER
admin@example.com	Library_access#2	USER, CURATOR, ADMIN
curator@example.com	Library_access#3	USER, CURATOR

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/oauth2-login-client* and *05-oauth2/oauth2-library-server*.

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

Chapter 6. Feedback

Please provide feedback for this workshop directly to organization of the Frankfurter Entwicklertag using the provided QR code.



If you have further 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](https://twitter.com/andifalk)
- LinkedIn: [andifalk](https://www.linkedin.com/in/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 2.1 Reference Guide](#)
- [Spring Security 4.x Reference Guide](#)
- [Spring Security 5.1.1 Reference Guide](#)
- [Legacy Spring Security OAuth Reference Guide](#)
- [Legacy Spring Security OAuth2 Boot Reference Guide](#)
- [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)