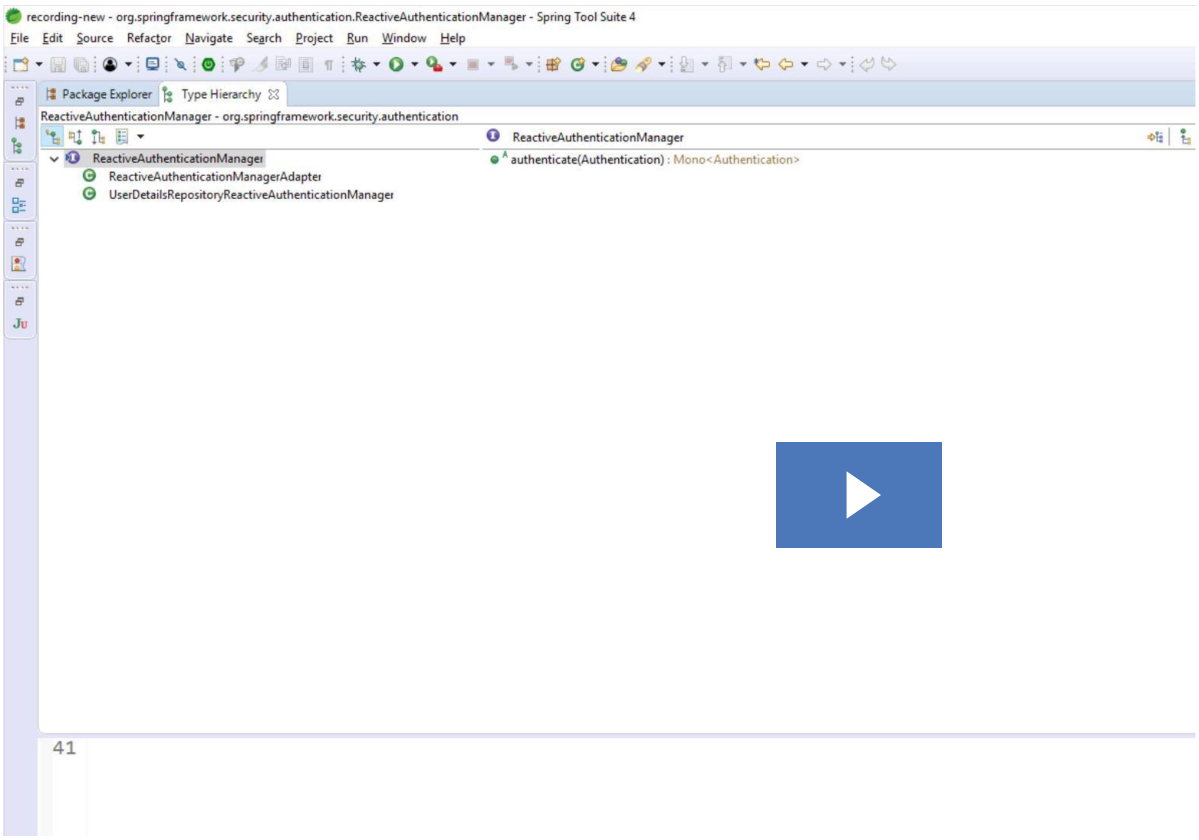
Lesson 1: A Basic Reactive Security Example (NEW)


recording-new - org.springframework.security.authentication.ReactiveAuthenticationManager - Spring Tool Suite 4
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Type Hierarchy
ReactiveAuthenticationManager - org.springframework.security.authentication
ReactiveAuthenticationManager
ReactiveAuthenticationManagerAdapter
UserDetailsRepositoryReactiveAuthenticationManager
ReactiveAuthenticationManager
authenticate(Authentication) : Mono<Authentication>
41

1. Goal

The goal of this lesson is to set up a basic security example for our unsecured WebFlux application. This is our standard User Management application that we've used throughout this course, but converted to a reactive application.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m16-lesson1-start

If you want to skip and see the complete implementation, feel free to jump ahead and import: m16-lesson1

Basic Security Configuration

Note that before starting this lesson, you should be familiar with the basics of WebFlux. Have a look at the *Rest With Spring - Reactive REST API* Module if you're a student there, or the WebFlux tutorials on the site.

Let's start up the application and notice it's no longer running on Tomcat, but Netty, as the log shows:

Netty started on port(s): 8080

Then, we can simply use the app:

http://localhost:8080/user

This is now entirely unsecured.

Let's add the *spring-boot-starter-security* dependency to our POM:

HTML
```html
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

This is the same security dependency as in a standard Web application. You'll also notice the configuration and annotations we'll be using here are corresponding to the standard Spring Security annotations.

Let's now create our security configuration. We'll use the corresponding annotation in the WebFlux world: *@EnableWebFluxSecurity* - similar to the standard *@EnableWebSecurity:*

JAVA

```java
@EnableWebFluxSecurity
public class LssSecurityConfig {
    // ...
}
```

Also similar to how in a servlet environment we had our authentication manager, in the reactive implementation we have the same thing. Here, the interface is called *ReactiveAuthenticationManager.*

We can see that, in the reactive world, the core concepts of the Security framework are the same. So we don't have to learn a whole new security paradigm.

Let's now start doing something practical and let's configure a couple of users:

JAVA

```java
@EnableWebFluxSecurity
public class LssSecurityConfig {
    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder().username("user").password("pass").roles("USER").build();
        UserDetails admin = User.withDefaultPasswordEncoder().username("admin").password("pass").roles("ADMIN").build();

        return new MapReactiveUserDetailsService(user, admin);
    }
}
```

The *MapReactiveUserDetailsService* is the default implementation we have available in a reactive application.

This implements the *ReactiveUserDetailsService* interface which is the same as the standard *UserDetailsInterface* with the only difference being that the method returns a *Publisher*.

We've also used the builders that Spring Security now provides to add 2 users. Note that we're using the *withDefaultPasswordEncoder()* here to build our quick example, but this is not suitable for a production application.

Let's now create a very basic security setup which simply secures all endpoints:

JAVA

```java
@EnableWebFluxSecurity
public class LssSecurityConfig {
    // ...

    @Bean
    public SecurityWebFilterChain securitygWebFilterChain(ServerHttpSecurity httpSecurity) {
        return httpSecurity
            .authorizeExchange()
            .anyExchange()
            .authenticated()
                .and()
            .httpBasic()
                .and()
            .csrf()
            .disable()
                .build();
    }
}
```

The *ServerHttpSecurity* is our entry point into our basic security configuration.

We've also used basic authentication and disabled CSRF (for now).

Let's make sure the new package is scanned:

JAVA

```java
@SpringBootApplication(scanBasePackages = {"com.baeldung.lss.web", "com.baeldung.lss.spring"})
public class LssApp1 {
    // ...
}
```

And let's now access our application at http://localhost:8080

Basic authentication is now enabled and we're getting prompted to authenticate.

We can also get more granular with our authorization configuration.

Let's restrict the */user/delete* endpoint to only users with the ADMIN role. We'll add the configuration for this before *anyExchange()*:

JAVA

```java
// ...
.authorizeExchange()
  .pathMatchers("/user/delete/*").hasRole("ADMIN")
.anyExchange()
.authenticated()
// ...
```

To test this, let's re-login with *user* and try to delete a user. We can see that *user* is not allowed to delete the resource because it doesn't have the ADMIN role.

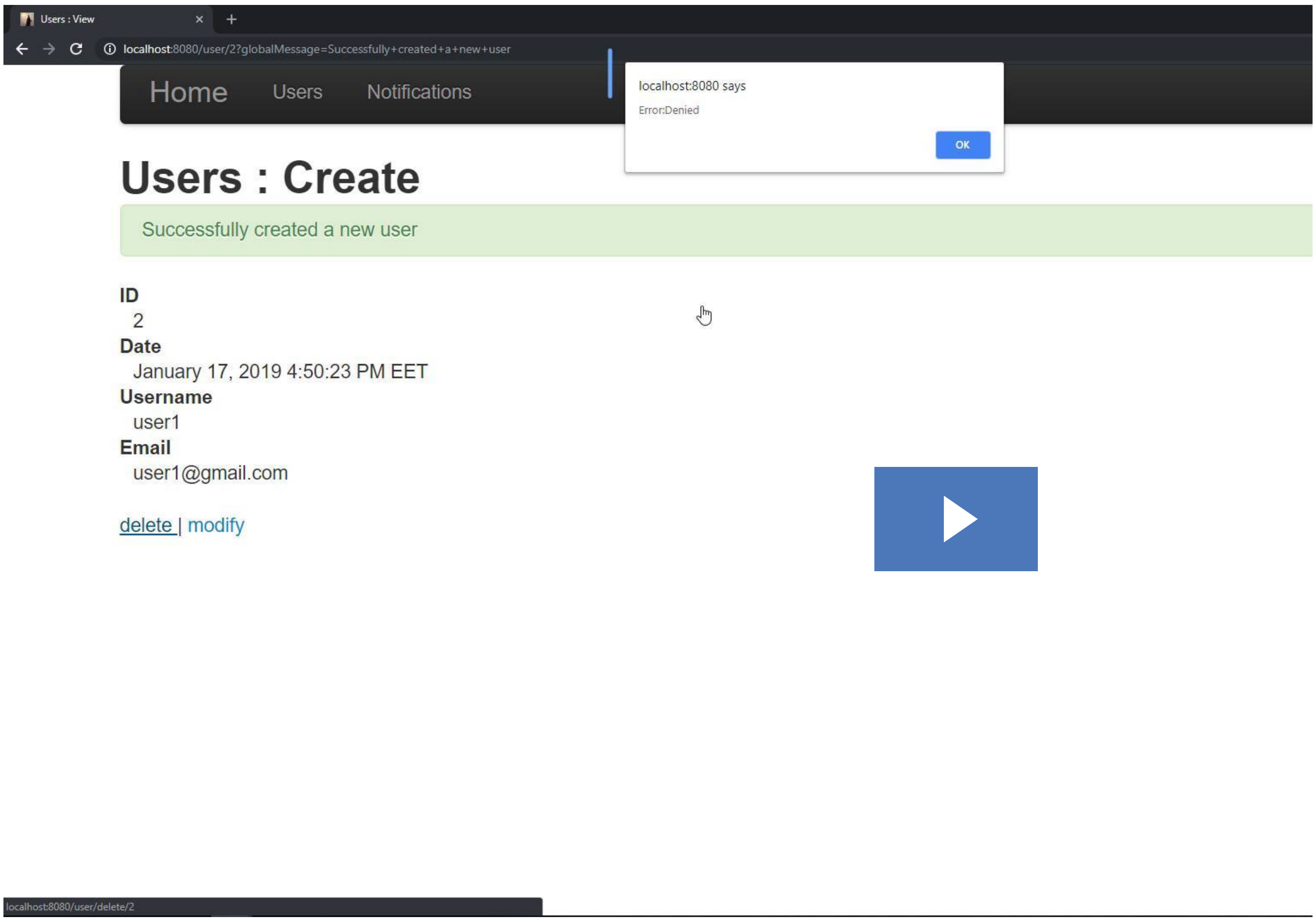
If we login with *admin* and delete the newly created user, the operation will be successful.

3. Resources

- WebFlux Security Configuration Reference

- Spring Security for Reactive Applications

Lesson 2: Reactive Method Security (NEW)


Users : View
localhost:8080/user/2?globalMessage=Successfully+created+a+new+user
Home
Users
Notifications
localhost:8080 says
Error:Denied
OK
delete | modify
localhost:8080/user/delete/2

Users : Create

Successfully created a new user

ID
2
Date
January 17, 2019 4:50:23 PM EET
Username
user1
Email
user1@gmail.com

1. Goal

In this lesson, we're going to configure method security for our WebFlux application.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m16-lesson2-start

If you want to skip and see the complete implementation, feel free to jump ahead and import: m16-lesson2

Security Configuration

In the previous lesson, we secured the */user/delete* endpoint using URL security. We'll now do the same thing, but using method level security.

First, we'll remove the URL security configuration.

Then, let's enable method security for the application using *@EnableReactiveMethodSecurity*:

JAVA
```java
@EnableWebFluxSecurity
@EnableReactiveMethodSecurity
public class LssSecurityConfig {
    // ...
}
```

This is the annotation corresponding to the standard *@EnableGlobalMethodSecurity.*

Now that this is enabled, let's go to the user DELETE operation in the *UserController* and add the security annotation *@PreAuthorize("hasRole('ADMIN')")*:

JAVA
```java
@Controller
@RequestMapping("/user")
public class UserController {

    // ...

    @DeleteMapping("delete/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    @PreAuthorize("hasRole('ADMIN')")
    public Mono<Void> delete(@PathVariable("id") Long id) {
        this.userRepository.deleteUser(id);
        return Mono.empty();
    }
}
```

We can see that this part is exactly the same as the servlet stack.

The operation is now restricted to users with *ADMIN* role.

Let's run the application and make sure everything works as expected: *http://localhost:8080/*

We'll login to the application with the *admin* account and check that everything works ok by creating a user and deleting it. If we log in with the *user* account and try to delete a user, we'll receive an Access Denied error.

Let's also write a simple test for the functionality that will check the security semantics of the operation without having to run the application and hit the endpoint from a browser.

We're going to use the reactive test support, more specifically the *StepVerifier* to call the *delete()* method:

JAVA
```java
@RunWith(SpringRunner.class)
@SpringBootTest(classes = LssApp2.class, webEnvironment = WebEnvironment.NONE)
public class SecurityIntegrationTest {

    @Autowired
    private UserController userApi;

    @Test
    public void whenDeleting_thenAccessDenied() {
        StepVerifier.create(this.userApi.delete(1l)).expectError(AccessDeniedException.class).verify();
    }
}
```

In this test method, we're verifying that, without authentication, we'll receive an *AccessDeniedException*.

Let's also test that a user with the ADMIN role will be able to consume this operation, by mocking a user with this role:

JAVA
```java
@Test
@WithMockUser(roles = "ADMIN")
public void givenAdminUser_whenDeleting_thenAccessDenied() {
    StepVerifier.create(this.userApi.delete(1l)).verifyComplete();
}
```

Then we can verify that the tests pass successfully.

Overall, it's important to note that the reactive configuration resembles and follows the servlet configuration. The Spring Security reactive support is very intuitive once you have a good handle on the servlet support.

3. Resources

- Reactive Method Security Reference

Lesson 3: WebFlux Form Login (NEW - Text Only)

1. Goal

This lesson contains an example of a WebFlux application secured using Form Login.

Enabling Form Login in a WebFlux application is very similar to the Form Login example in a standard Web application that we've seen in *Module 1 - Building a Form Login* lesson.

This, combined with the previous lessons on Reactive Security in this module, contain all the necessary background to build our example.

For this reason, we won't go into too much detail here again, but it's useful to have this example as a reference implementation of Form Login security in a Spring WebFlux application.

2. Lesson Notes

The relevant module you need to import for this lesson is: m16-lesson3

Spring WebFlux - Form Login

Let's have a look at the *LssSecurityConfig* class:

```java
@EnableWebFluxSecurity
@EnableReactiveMethodSecurity
public class LssSecurityConfig {

    // ...

    @Bean
    SecurityWebFilterChain springWebFilterChain(ServerHttpSecurity httpSecurity) throws Exception {
        return httpSecurity
                .authorizeExchange()
                .pathMatchers("/user/delete/*").hasRole("ADMIN")
                .pathMatchers("/favicon.ico").permitAll()
                .anyExchange()
                .authenticated()
                    .and()
                .formLogin()
                    .and()
                .csrf()
                .disable()
                    .build();
    }

}
```

Here, we've added the *formLogin()* method in the WebFlux security configuration we've already seen.

3. References

- Spring Security WebFlux Form Login