

# Spring Security 5 for Reactive Applications

Last modified: September 2, 2019

| by [baeldung](#)

Reactive

Spring Security

Spring Web

Spring 5

Spring Security 5

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

**>> CHECK OUT THE COURSE**

---

## 1. Introduction

In this article, we'll explore new features of the [Spring Security 5](#) framework for securing reactive applications. This release is aligned with Spring 5 and Spring Boot 2.

In this article, we won't go into details about the reactive applications themselves, which is a new feature of the Spring 5 framework. Be sure to check out the article [Intro to Reactor Core](#) for more details.

## 2. Maven Setup

We'll use Spring Boot starters to bootstrap our project together with all required dependencies.

The basic setup requires a parent declaration, web starter, and security starter dependencies. We'll also need the Spring Security test framework:

```
1 | <parent>
```

```
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-parent</artifactId>
4     <version>2.0.0.RELEASE</version>
5     <relativePath/>
6 </parent>
7
8 <dependencies>
9     <dependency>
10         <groupId>org.springframework.boot</groupId>
11         <artifactId>spring-boot-starter-web</artifactId>
12     </dependency>
13     <dependency>
14         <groupId>org.springframework.boot</groupId>
15         <artifactId>spring-boot-starter-security</artifactId>
16     </dependency>
17     <dependency>
18         <groupId>org.springframework.security</groupId>
19         <artifactId>spring-security-test</artifactId>
20         <scope>test</scope>
21     </dependency>
22 </dependencies>
```

We can check out the current version of Spring Boot security starter [over at Maven Central](#).

### 3. Project Setup

### 3.1. Bootstrapping the Reactive Application

We won't use the standard *@SpringBootApplication* configuration but instead, configure a Netty-based web server. **Netty is an asynchronous NIO-based framework which is a good foundation for reactive applications.**

The *@EnableWebFlux* annotation enables the standard Spring Web Reactive configuration for the application:

```
1  @ComponentScan(basePackages = {"com.baeldung.security"})
2  @EnableWebFlux
3  public class SpringSecurity5Application {
4
5      public static void main(String[] args) {
6          try (AnnotationConfigApplicationContext context
7              = new AnnotationConfigApplicationContext(
8                  SpringSecurity5Application.class)) {
9
10             context.getBean(NettyContext.class).onClose().block();
11         }
12     }
```

Here, we create a new application context and wait for Netty to shut down by calling *.onClose().block()* chain on the Netty context.

After Netty is shut down, the context will be automatically closed using the *try-with-resources* block.

We'll also need to create a Netty-based HTTP server, a handler for the HTTP requests, and the adapter between the server and the handler:

```
1 | @Bean
2 | public NettyContext nettyContext(ApplicationContext context) {
3 |     HttpHandler handler = WebHttpHandlerBuilder
4 |         .applicationContext(context).build();
5 |     ReactorHttpHandlerAdapter adapter
6 |         = new ReactorHttpHandlerAdapter(handler);
7 |     HttpServer httpServer = HttpServer.create("localhost", 8080);
8 |     return httpServer.newHandler(adapter).block();
9 | }
```

## 3.2. Spring Security Configuration Class

For our basic Spring Security configuration, we'll create a configuration class – *SecurityConfig*.

To enable WebFlux support in Spring Security 5, we only need to specify the *@EnableWebFluxSecurity* annotation:

```
1 | @EnableWebFluxSecurity
2 | public class SecurityConfig {
3 |     // ...
```

Now we can take advantage of the class *ServerHttpSecurity* to build our security configuration.

**This class is a new feature of Spring 5.** It's similar to *HttpSecurity* builder, but it's only enabled for WebFlux applications.

The *ServerHttpSecurity* is already preconfigured with some sane defaults, so we could skip this configuration completely. But for starters, we'll provide the following minimal config:

```
1  @Bean
2  public SecurityWebFilterChain securityWebFilterChain(
3      ServerHttpSecurity http) {
4      return http.authorizeExchange()
5          .anyExchange().authenticated()
6          .and().build();
7  }
```

Also, we'll need a user details service. Spring Security provides us with a convenient mock user builder and an in-memory implementation of the user details service:

```
1  @Bean
2  public MapReactiveUserDetailsService userDetailsService() {
3      UserDetails user = User
4          .withUsername("user")
5          .password(passwordEncoder().encode("password"))
6          .roles("USER")
7          .build();
8      return new MapReactiveUserDetailsService(user);
9  }
```

Since we're in reactive land, the user details service should also be reactive. If we check out the *ReactiveUserDetailsService* interface, **we'll see that its *findByUsername* method actually returns a *Mono* publisher:**

```
1 public interface ReactiveUserDetailsService {  
2  
3     Mono<UserDetails> findByUsername(String username);  
4 }
```

Now we can run our application and observe a regular HTTP basic authentication form.

## 4. Styled Login Form

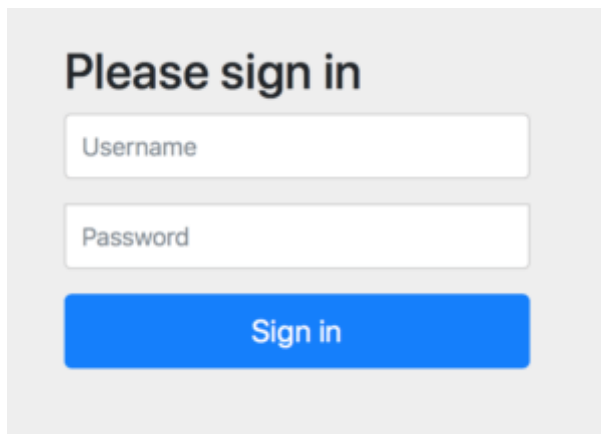


A small but striking improvement in Spring Security 5 is a new styled login form which uses the Bootstrap 4 CSS framework. The stylesheets in the login form link to CDN, so we'll only see the improvement when connected to the Internet.

To use the new login form, let's add the corresponding *formLogin()* builder method to the *ServerHttpSecurity* builder:

```
1 public SecurityWebFilterChain securityWebFilterChain(  
2     ServerHttpSecurity http) {  
3     return http.authorizeExchange()  
4         .anyExchange().authenticated()  
5         .and().formLogin()  
6         .and().build();  
7 }
```

If we now open the main page of the application, we'll see that it looks much better than the default form we're used to since previous versions of Spring Security:



The image shows a login form with a light gray background. At the top, it says "Please sign in" in bold black text. Below this, there are two input fields: "Username" and "Password", both with light gray borders and placeholder text. At the bottom, there is a blue button with the text "Sign in" in white.

**Note that this is not a production-ready form, but it's a good bootstrap of our application.**

If we now log in and then go to the <http://localhost:8080/logout> URL, we'll see the logout confirmation form, which is also styled.

## 5. Reactive Controller Security

To see something behind the authentication form, let's implement a simple reactive controller that greets the user:

```
1  @RestController
2  public class GreetController {
3
4      @GetMapping("/")
5      public Mono<String> greet(Mono<Principal> principal) {
6          return principal
7              .map(Principal::getName)
8              .map(name -> String.format("Hello, %s", name));
9      }
10
11 }
```

After logging in, we'll see the greeting. Let's add another reactive handler that would be accessible by admin only:

```
1  @GetMapping("/admin")
2  public Mono<String> greetAdmin(Mono<Principal> principal) {
3      return principal
4          .map(Principal::getName)
```

```
5 |         .map(name -> String.format("Admin access: %s", name));  
6 |     }
```

Now let's create a second user with the role *ADMIN* in our user details service:

```
1 | UserDetails admin = User.withDefaultPasswordEncoder()  
2 |     .username("admin")  
3 |     .password("password")  
4 |     .roles("ADMIN")  
5 |     .build();
```

We can now add a matcher rule for the admin URL that requires the user to have the *ROLE\_ADMIN* authority.

**Note that we have to put matchers before the *.anyExchange()* chain call.** This call applies to all other URLs which were not yet covered by other matchers:

```
1 | return http.authorizeExchange()  
2 |     .pathMatchers("/admin").hasAuthority("ROLE_ADMIN")
```

```
3 | .anyExchange().authenticated()
4 | .and().formLogin()
5 | .and().build();
```

If we now log in with *user* or *admin*, we'll see that they both observe initial greeting, as we've made it accessible for all authenticated users.

**But only the *admin* user can go to the <http://localhost:8080/admin> URL and see her greeting.**

## 6. Reactive Method Security

We've seen how we can secure the URLs, but what about methods?

To enable method-based security for reactive methods, we only need to add the `@EnableReactiveMethodSecurity` annotation to our `SecurityConfig` class:

```
1 | @EnableWebFluxSecurity
2 | @EnableReactiveMethodSecurity
3 | public class SecurityConfig {
4 |     // ...
5 | }
```

Now let's create a reactive greeting service with the following content:

```
1 | @Service
2 | public class GreetService {
3 |
4 |     public Mono<String> greet() {
```

```
5 |         return Mono.just("Hello from service!");
6 |     }
7 | }
```

We can inject it into the controller, go to <http://localhost:8080/greetService> and see that it actually works:

```
1 | @RestController
2 | public class GreetController {
3 |
4 |     private GreetService greetService
5 |
6 |     @GetMapping("/greetService")
7 |     public Mono<String> greetService() {
8 |         return greetService.greet();
9 |     }
10 |
11 |     // standard constructors...
12 | }
```

But if we now add the `@PreAuthorize` annotation on the service method with the `ADMIN` role, then the greet service URL won't be accessible to a regular user:

```
1 | @Service
2 | public class GreetService {
3 |
4 |     @PreAuthorize("hasRole('ADMIN')")
```

```
5 | public Mono<String> greet() {  
6 |     // ...  
7 | }
```

## 7. Mocking Users in Tests

Let's check out how easy it is to test our reactive Spring application.

First, we'll create a test with an injected application context:

```
1 | @ContextConfiguration(classes = SpringSecurity5Application.class)  
2 | public class SecurityTest {  
3 |  
4 |     @Autowired  
5 |     ApplicationContext context;  
6 |  
7 |     // ...  
8 | }
```

Now we'll set up a simple reactive web test client, which is a feature of the Spring 5 test framework:

```
1 | @Before  
2 | public void setup() {  
3 |     this.rest = WebClient  
4 |         .bindToApplicationContext(this.context)  
5 |         .configureClient()  
6 |         .build();  
7 | }
```

This allows us to quickly check that the unauthorized user is redirected from the main page of our application to the login page:

```
1  @Test
2  public void whenNoCredentials_thenRedirectToLogin() {
3      this.rest.get()
4          .uri("/")
5          .exchange()
6          .expectStatus().is3xxRedirection();
7  }
```

**If we now add the `@MockWithUser` annotation to a test method, we can provide an authenticated user for this method.**

The login and password of this user would be *user* and *password* respectively, and the role is *USER*. This, of course, can all be configured with the `@MockWithUser` annotation parameters.

Now we can check that the authorized user sees the greeting:

```
1  @Test
2  @WithMockUser
3  public void whenHasCredentials_thenSeesGreeting() {
4      this.rest.get()
5          .uri("/")
6          .exchange()
7          .expectStatus().isOk()
8          .expectBody(String.class).isEqualTo("Hello, user");
9  }
```

The `@WithMockUser` annotation is available since Spring Security 4. However, in Spring Security 5 it was also updated to cover reactive endpoints and methods.

## 8. Conclusion

In this tutorial, we've discovered new features of the upcoming Spring Security 5 release, especially in the reactive programming arena.

As always, the source code for the article is available [over on GitHub](#).

**I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:**

**>> CHECK OUT THE COURSE**

▲ newest ▲ **oldest** ▲ most voted





flo



Thank you for this example.

I think create a new NettyContext is not necessary if you use spring-boot-starter-webflux instead of spring-boot-starter-web. Use standard application configuration with @SpringBootApplication and @EnableWebFluxSecurity working well.

+ 2 -

🕒 1 year ago



Loredana Crusoveanu



Hey Florian,

From what I see, it's still using Tomcat by default, without the explicit Netty configuration.

+ 0 -

🕒 1 year ago

Comments are closed on this article!



CATEGORIES

SERIES

ABOUT

[SPRING](#)

[REST](#)

[JAVA](#)

[SECURITY](#)

[PERSISTENCE](#)

[JACKSON](#)

[HTTP CLIENT-SIDE](#)

[KOTLIN](#)

[JAVA "BACK TO BASICS" TUTORIAL](#)

[JACKSON JSON TUTORIAL](#)

[HTTPCLIENT 4 TUTORIAL](#)

[REST WITH SPRING TUTORIAL](#)

[SPRING PERSISTENCE TUTORIAL](#)

[SECURITY WITH SPRING](#)

[ABOUT BAELDUNG](#)

[THE COURSES](#)

[CONSULTING WORK](#)

[META BAELDUNG](#)

[THE FULL ARCHIVE](#)

[WRITE FOR BAELDUNG](#)

[EDITORS](#)

[OUR PARTNERS](#)

[ADVERTISE ON BAELDUNG](#)

[TERMS OF SERVICE](#) | [PRIVACY POLICY](#) | [COMPANY INFO](#) | [CONTACT](#)