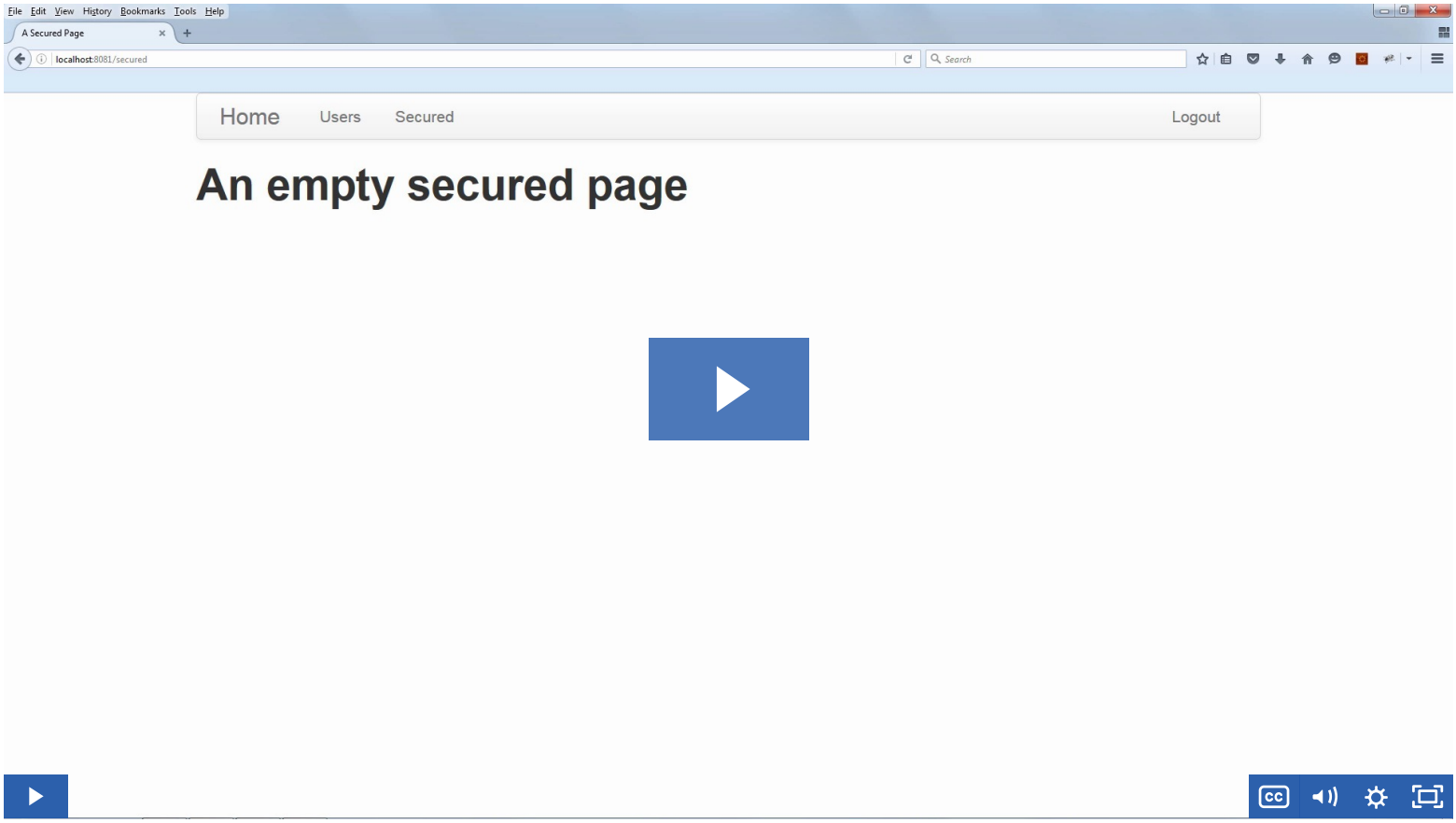
Lesson 1: By URL Authorization with Expressions



1. Main Goals

The focus of this lesson is to understand the full range of Spring Security expressions we have at our disposal.

2. Lesson Notes

If you're using the git repository to get the project - you should be using the module5 branch.

The relevant module you need to import when you're starting with this lesson is: m5-lesson1

The credentials used in the code of this lesson are:
user/pass (in-memory)
admin/pass (in-memory) (has ADMIN role)

Throughout this lesson, we're going to be setting up different types of authorization on our /secured page.

We're going to start with the original examples from previous lessons:

```
.antMatchers("/secured").access("hasRole('USER')")
.antMatchers("/secured").access("hasAuthority('ROLE_USER')")
```

hasIpAddress

Let's do a quick example first:

```
.antMatchers("/secured").hasIpAddress("192.168.1.0/24")
```

So we're using an IP which we do not have when deploying on localhost - to see how accessing the page now results in a 403 Forbidden.

Let's now change the value it to localhost (IPv6 enabled):

```
.antMatchers("/secured").hasIpAddress("::1")
```

anonymous

Again, let's start with a quick example - first using the API and then doing the same with expressions:

```
.antMatchers("/secured").anonymous()
.antMatchers("/secured").access("isAnonymous()")
```

We can easily test that the new required authorization works by simply first accessing the page logged in - and seeing the 403, and then removing the cookie (thus becoming unauthenticated) and accessing the page again, now successfully.

working with the request object directly

The example here is quite interesting:

```
.antMatchers("/secured").access("request.method == 'GET'")
```

This will basically restrict access to that page to only GET requests.

Before moving on from this example, remember that this syntax is quite flexible:

```
.antMatchers("/secured").access("request.method != 'POST'")
```

This will allow **anything but** POST requests.

negating an expression - not

```
.antMatchers("/secured").not().access("hasIpAddress('::1')")
```

logical concatenation of expressions

We can concatenate multiple expressions to form even more powerful and flexible expressions.

Here's a couple of quick example of this new syntax:

```
.antMatchers("/secured").access("hasRole('ROLE_ADMIN') and principal.username == 'user'")
.antMatchers("/secured").access("hasRole('ROLE_ADMIN') or principal.username == 'user'")
```

Finally, it's important to understand that the order of URL definitions in the configuration is important. The more specific URLs authorization configs need to come first and and the more general ones need to be later - so that the first ones get applied.

A quick and simple way to test this is to move any of these example expressions bellow the generic:

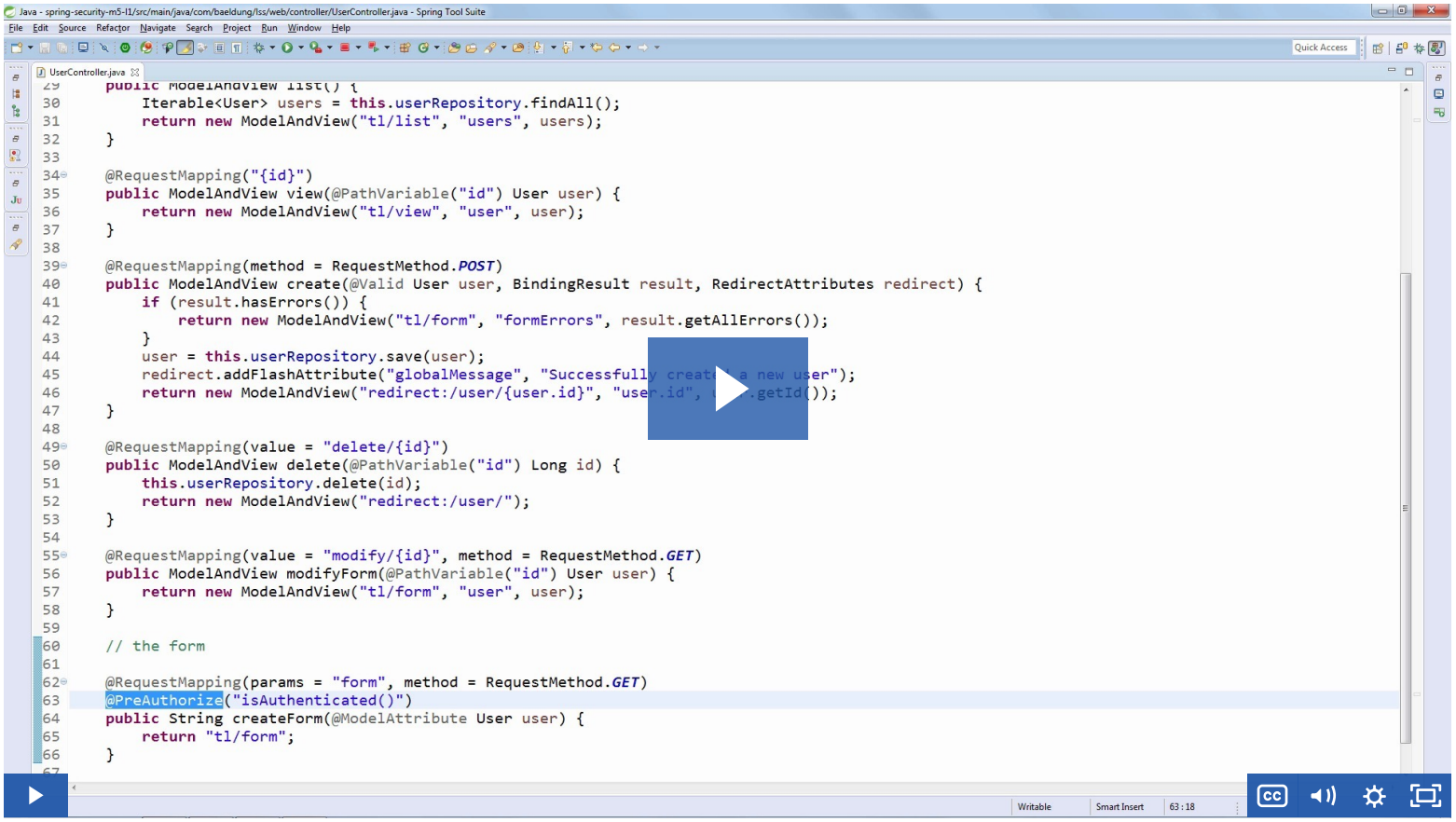
```
.anyRequest().authenticated()
```

And evaluate the results.

3. Resources

- Expression-Based Access Control in the Spring Reference

Lesson 2: On-method Authorization with Expressions



1. Main Goal

The focus of the lesson is to introduce method level authorization with Spring Expressions.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m5-lesson1

Let's start by enabling method security:

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
```

Now, for demo purposes, we're going to disable the URL level authorization and allow everything:

```
.anyRequest().permitAll()
```

At this point everything's now accessible (including the create form which we'll use as the main example in this lesson).

Next, let's make use of the *@PreAuthorize* annotation and secure our first controller method:

```
@PreAuthorize("isAuthenticated()")
```

We can easily test this by accessing the create form - first logged in, and then logged out.

Next, we're going to use another expression in the *@PreAuthorize* annotation (on the same *createForm* method):

```
@PreAuthorize("hasRole('ADMIN')")
```

Finally, let's try out another expression:

```
@PreAuthorize("principal.username=='user'")
```

We are going to also briefly show the *@Secure* annotation here as well.

Note that the main difference between *@PreAuthorized* and *@Secure* is that *@Secure* has no support for using expression.

The syntax that it does accept is very simple:

```
@Secured("ROLE_ADMIN")
```

A final note here is that we only disabled URL authorization for demo purposes, and you can definitely combine URL and method level authorization in production.

3. Resources

- Method Security in the Official Reference

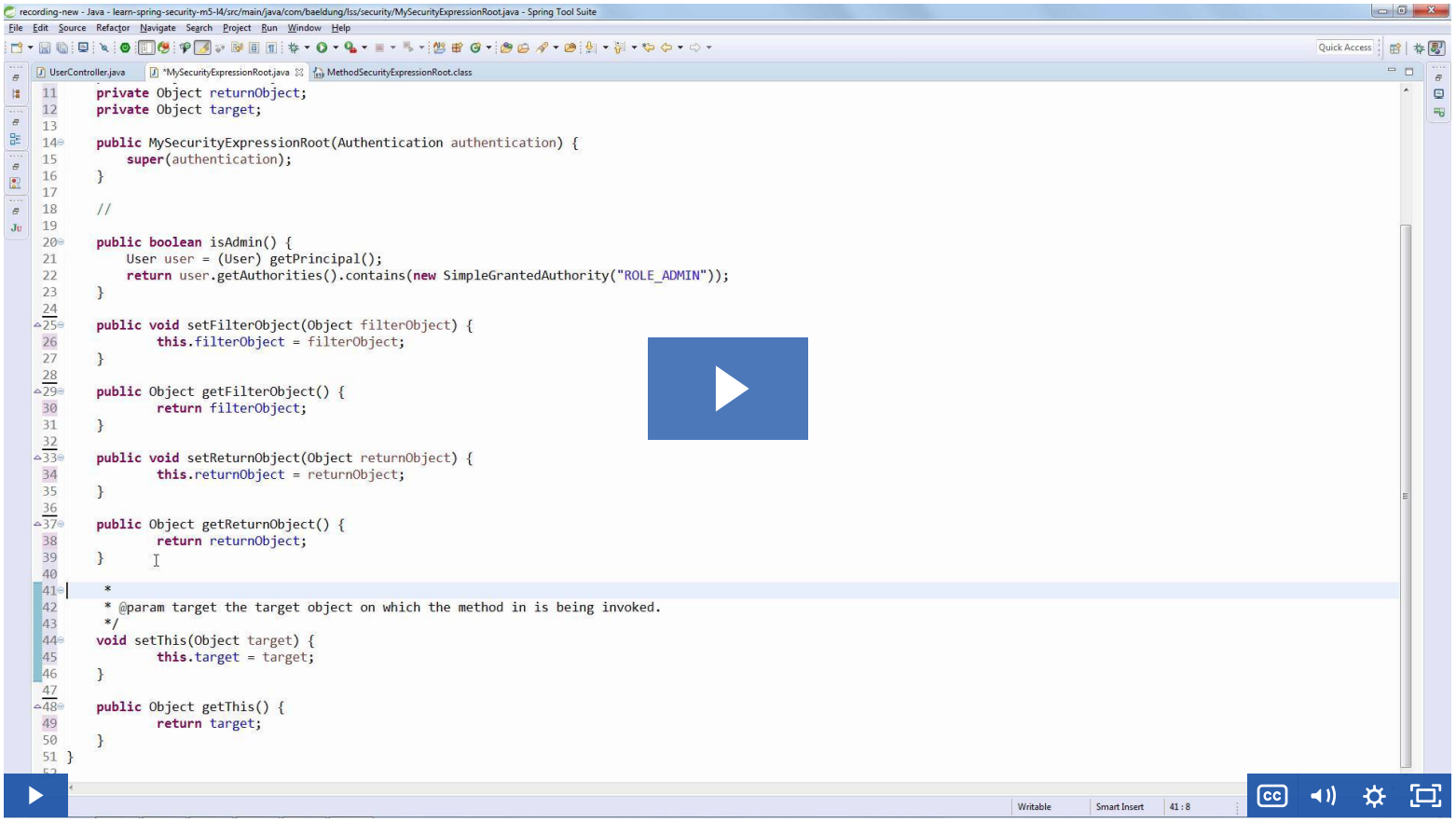
Lesson 3: In-page URL Authorization with Expressions (same as Module 4 - Lesson 4)

This lesson is the same as Lesson 4 from Module 4.

The reason it's included under Module 5 as well is because this Module 5 is focused on Spring Expressions, so the lesson naturally fits here as well.

So - check out Module 4 - Lesson 4 if you haven't already.

Lesson 4: Programmatic Expressions and a custom PermissionEvaluator (NEW)



1. Goal

The goal of this lesson is to introduce and define a new security expression we can use along with the default expressions in the framework.

2. Lesson Notes

2.1. A New Security Expression

Security expressions are a core part of Spring Security, and we do have a number of them available in the framework.

But, sometimes it can still be quite useful to be able to define our own - so that's what we're going to do in this lesson - define a custom security expression.

We're going to define a simple expression - *isAdmin()*.

Here's how we'd like to use it - in the *UserController*:

```
@PreAuthorize("isAdmin()")
public ModelAndView list() {
```

OK, so let's get going.

2.2. A New *SecurityExpressionRoot*

First, we're going to define our own SecurityExpressionRoot:

```
public class MySecurityExpressionRoot extends SecurityExpressionRoot {
    public MySecurityExpressionRoot(Authentication authentication) {
        super(authentication);
    }
}
```

Next, we're going to simply implement our new expression as a method here:

```
public boolean isAdmin() {
    return false;
}
```

But of course we don't want to simply return false here. What we want is to look at the currently authenticated principal and determine if they're an admin or not:

```
User principal = (User) getPrincipal();
```

Here, we're simply using this convenient getPrincipal() API and we're casting to a User - which is the default User implementation in Spring Security.

Now, one simple option to determine if the user is an admin or not would be to look at its authorities:

```
return principal.getAuthorities().contains(new SimpleGrantedAuthority("ROLE_ADMIN"));
```

As you can see, this is a very simple implementation of determining if the currently authenticated principal is an admin in our system or not.

Of course the way to determine that will vary from one system to another. And given that we have access to the full principal, we have full flexibility over how we implement that.

Next, because we're dealing with method level security here, we also need to implement filtering. Let's first implement the relevant interface - *MethodSecurityExpressionOperations*.

And, with the new API, we're going to simple make use of the default implementation we already have available in the *MethodSecurityExpressionRoot*.

As a quick note - it would have been ideal to extend this class directly, so that we don't have to copy-paste the implementation - but, for now, that's not possible (the class is package private).

2.3. The New Security Config

Now, we finally need to wire in the new security expression root we just defined:

```
public class CustomMethodSecurityExpressionHandler extends DefaultMethodSecurityExpressionHandler {
}
```

And provide a *createSecurityExpressionRoot* implementation.

And, again, using the default impl is a great starting point; we'll only need to swap out the existing root expression with our own implementation and restore some of the defaults.

Finally, let’s wire that into our security configuration.

We’re going to define a separate class for our method configuration - so that we’re able to extend the base class provided by the framework:

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class CustomMethodSecurityConfig extends GlobalMethodSecurityConfiguration {
    //
}
```

And now, we can finally wire in and use our expression handler:

```
@Override
protected MethodSecurityExpressionHandler createExpressionHandler() {
    CustomMethodSecurityExpressionHandler expressionHandler = new CustomMethodSecurityExpressionHandler();
    return expressionHandler;
}
```

And that's it - our new security expression - the one we used in the controller at the very beginning - now works as expected.

2.4. New Live Tests

With everything set up, let’s now modify the live test to no longer expect a 200 OK.

The test is now going to expect a 403 Forbidden, to make sure the new annotation works.

Let's also add a new test to make sure the admin is actually able to access this secured operations:

```
@Test
public void givenAdmin_whenGetAllUsers_thenOK() {
    Response response = givenAuth("admin", "pass").get(APP_ROOT + "/user");
    assertEquals(200, response.getStatusCode());
}
```

And there we have it - we’re good to go.

3. Resources

- A Custom Security Expression with Spring Security