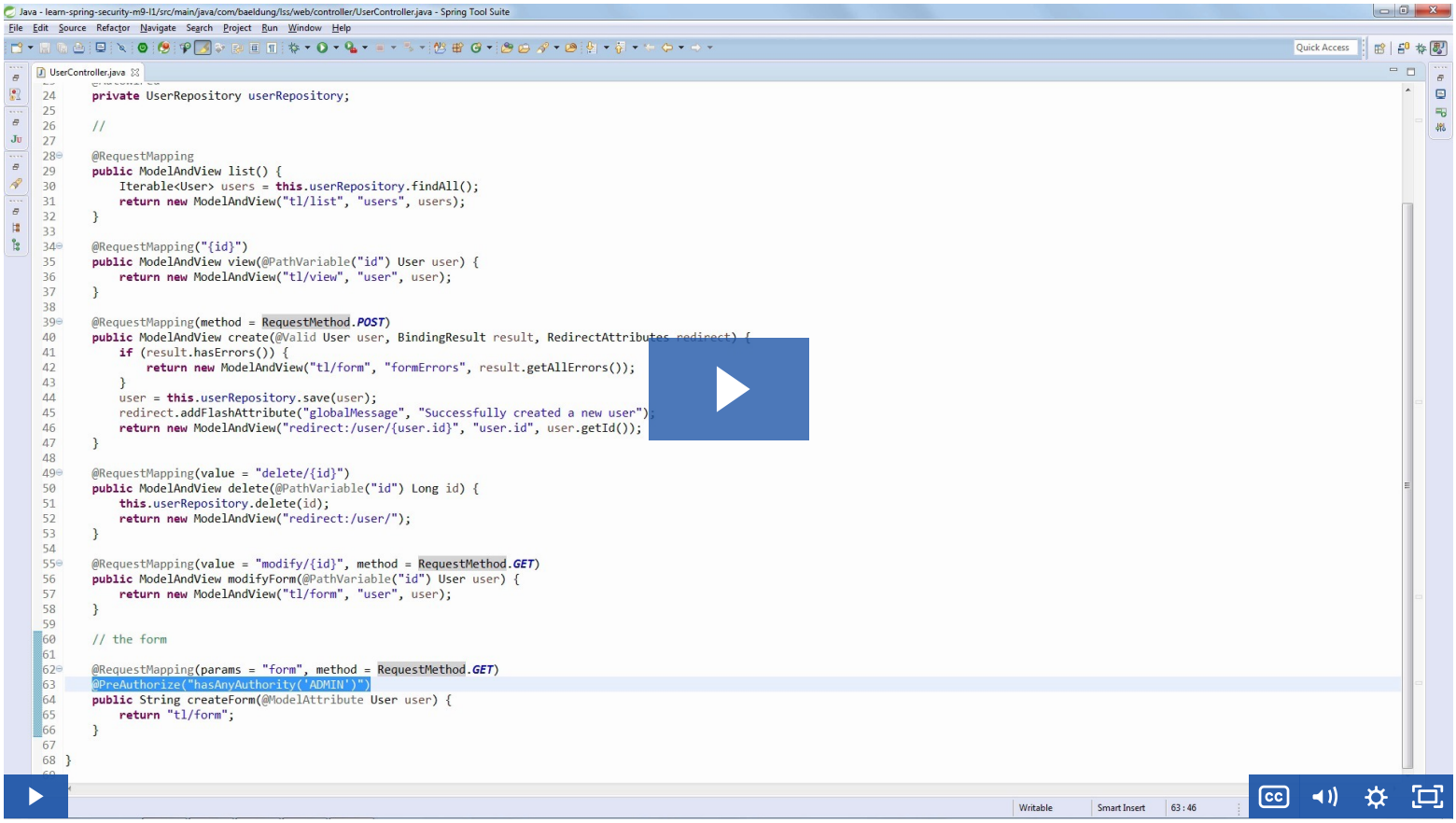
Lesson 1: How Authorization Works



1. Goals

The goal of this lesson is to guide you through the actual implementation details of the authorization process in Spring Security.

2. Lesson Notes

As you're using the git repository to get the project - you should be on the module9 branch.

Or, on the corresponding Spring Boot 2 based branch: module9-spring-boot-2.

The relevant module you need to import when you're starting with this lesson is: m9-lesson1.

The credentials used in the code of the lesson are:

user/pass (in-memory)

admin/pass (in-memory) (has the *ADMIN* role)

2.1. The Authorization Actors

The *AccessDecisionManager* has a simple API - with *decide* as the main entry point. It receives all the relevant info to make the authorization decision.

We have 3 implementation of available:

- *AffirmativeBased* - any affirmative voter will grant access
- *ConsensusBased* - a majority of affirmative voters will grant access
- *UnanimousBased* - all affirmative voters are required to grant access

By by default, the authorization strategy uses:

- an *AffirmativeBased AccessDecisionManager*
- with a *RoleVoter* and an *AuthenticatedVoter*

So we can now see the other major component here - **the voter**.

We have an *AccessDecisionVoter* and quite a few implementations - one for each type of authorization we can set up:

- Role (and one for hierarchical Roles)
- Authenticated
- Web Expressions
- ACL
- etc (a few others)

2.2. Debugging the Code

We're going to start with a simple setup - our config allows everything at the URL level and uses method security to secure the displaying of the create user form.

So, let's log in with user (which doesn't have enough permissions) - and let's follow the logic.

First - the *FilterSecurityInterceptor* is the last in the filter chain - this runs and calls the *AccessDecisionManager - decide*.

This simply goes over the voters and - if any of the voters deny access - then the *AccessDeniedException* is thrown.

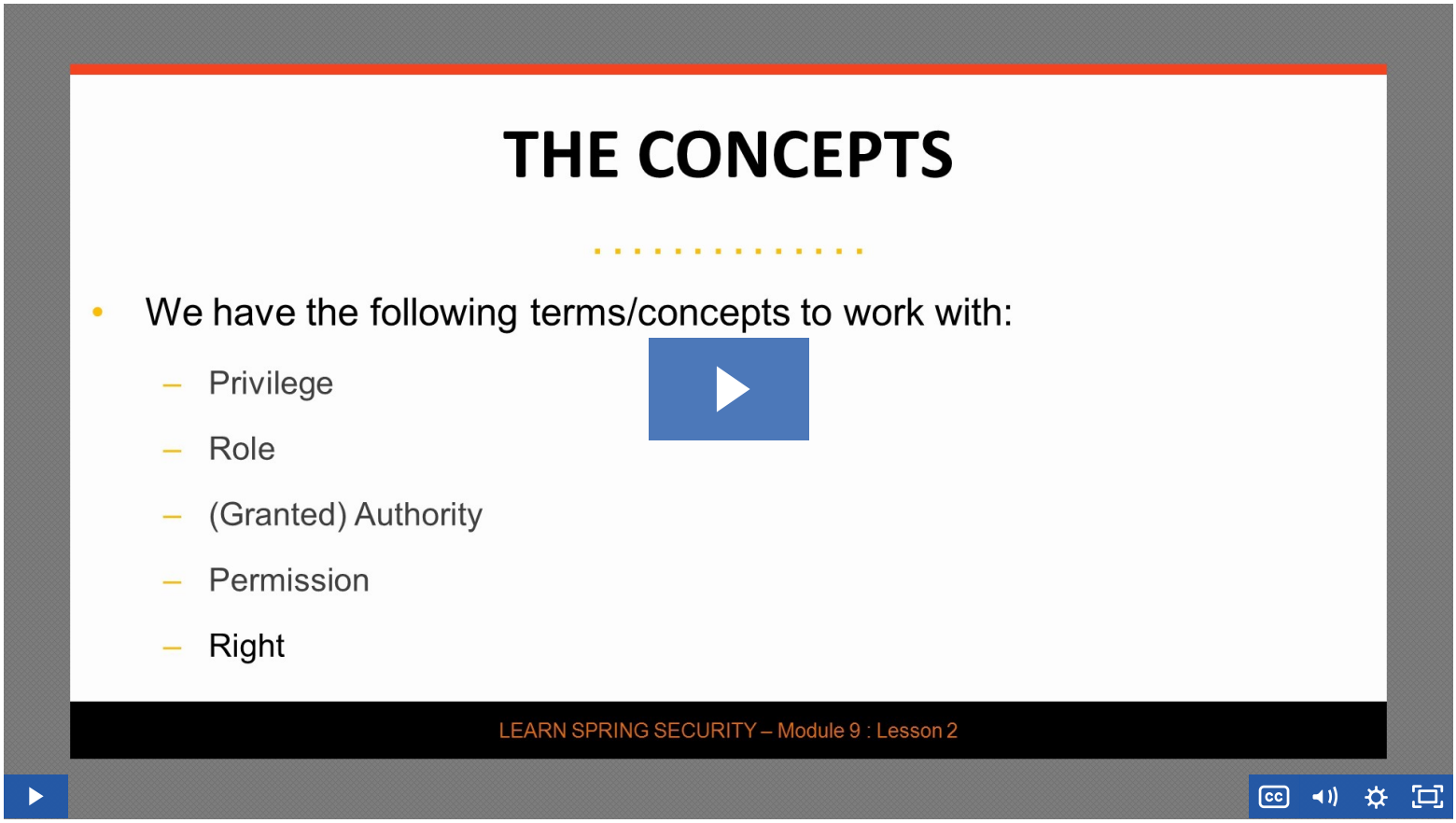
Otherwise, **access is granted**.

3. Resources

- The Default AccessDecisionManager in the Official Reference

- Pre-Invocation Handling in the Official Reference

Lesson 2: The Topology of Roles and Privileges - Part 1



1. Goals

The goal of the first part of this lesson is to illustrate a production-ready approach to structuring your roles and privileges.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m9-lesson2-start

If you want to skip and see the complete implementation, feel free to jump ahead and import: m9-lesson2

2.1. Why a Flat Topology isn't Enough

The Spring Security docs tend to talk about a simplistic approach, hardcoding roles like this:

```
@PreAuthorize("hasRole('doctor')")
public Patient getPatientRecords() { ... }
```

But, what if we need nurses to now have access to the patient records as well? We'll have to do:

```
@PreAuthorize("hasRole('doctor') or hasRole('nurse')")
public Patient getPatientRecords(...) { ... }
```

But what if now, the hospital administrator needs the same kind of acces. We'll have to do this:

```
@PreAuthorize("hasRole('doctor') or hasRole('nurse') or hasRole('hospitalAministrator')")
public Patient getPatientRecords() { ... }
```

So we can start seeing how that's not going to work.

2.2. The Authorization Concepts

Before we look into a solution, let's first look at **the core authorization concepts**:

- the Privilege
- the Role
- the (Granted) Authority
- the Permission
- the Right

The Spring Security docs use some of these interchangeably and with not a lot of rigor.

The first thing we need to do is to clearly decide and define what these mean for our system.

We are going to use Privilege to represent **a granular, low level capability in the system**. For example:

- *can delete a Resource 1 - that's a Privilege*
- *can update Resource 1 - is another Privilege*
- *can delete Resource 2 - is yet another, different Privilege*

Now, we're going to define Permission, Right and (Granted) Authority to mean the same thing.

And, since these will mean the exact same thing - we won't use them and we'll just use Privilege. With one small exception - sometimes we'll need to use Granted Authority because it's the name of the class in Spring Security.

Now that we've defined Privilege - as the main concept, and Permission, Right and Authority as secondary names for the same concept - **let's now look at the Role**.

We'll define Role as a collection of Privileges, and **a higher level concept that's user facing**. For example:

- Doctor
- Patient
- Hospital Administrator

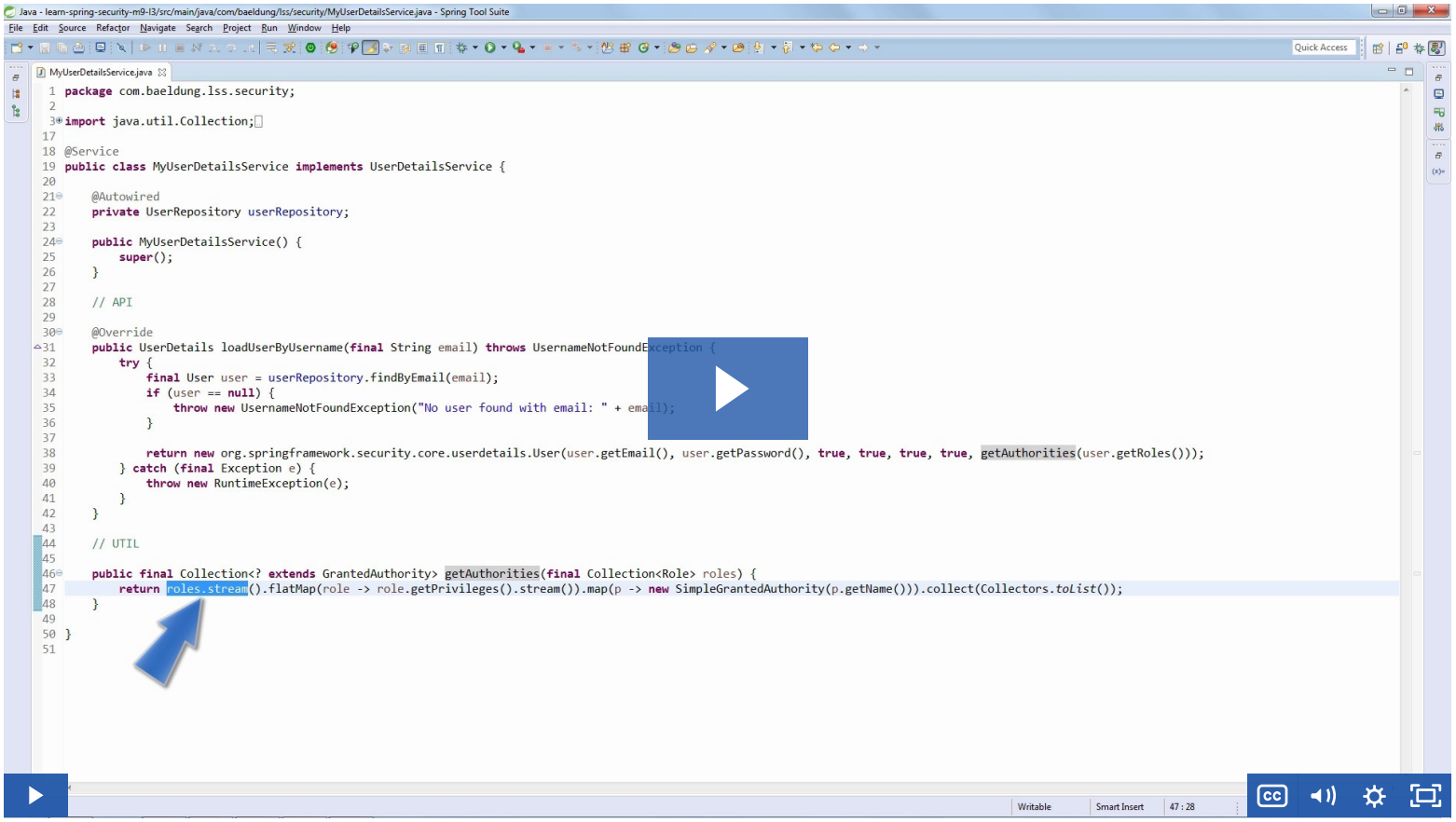
One quick side-note - the point of this topology is not to present it as THE only possible way; it's to present one possible way to think about these terms and concepts and of architecting them. Basically, the point is to show you HOW to think about these - for your own system.

If you adopt this topology - good, it's a solid way to go. But if you diverge from it - that’s perfectly fine as well. What’s important is that you make a decision, define the terms and use them consistently.

3. Resources

- Spring Security – Roles and Privileges

Lesson 2: The Topology of Roles and Privileges - Part 2



1. Goals

The goal of the second part of this lesson is to show the implementation of the aspects discussed in part 1.

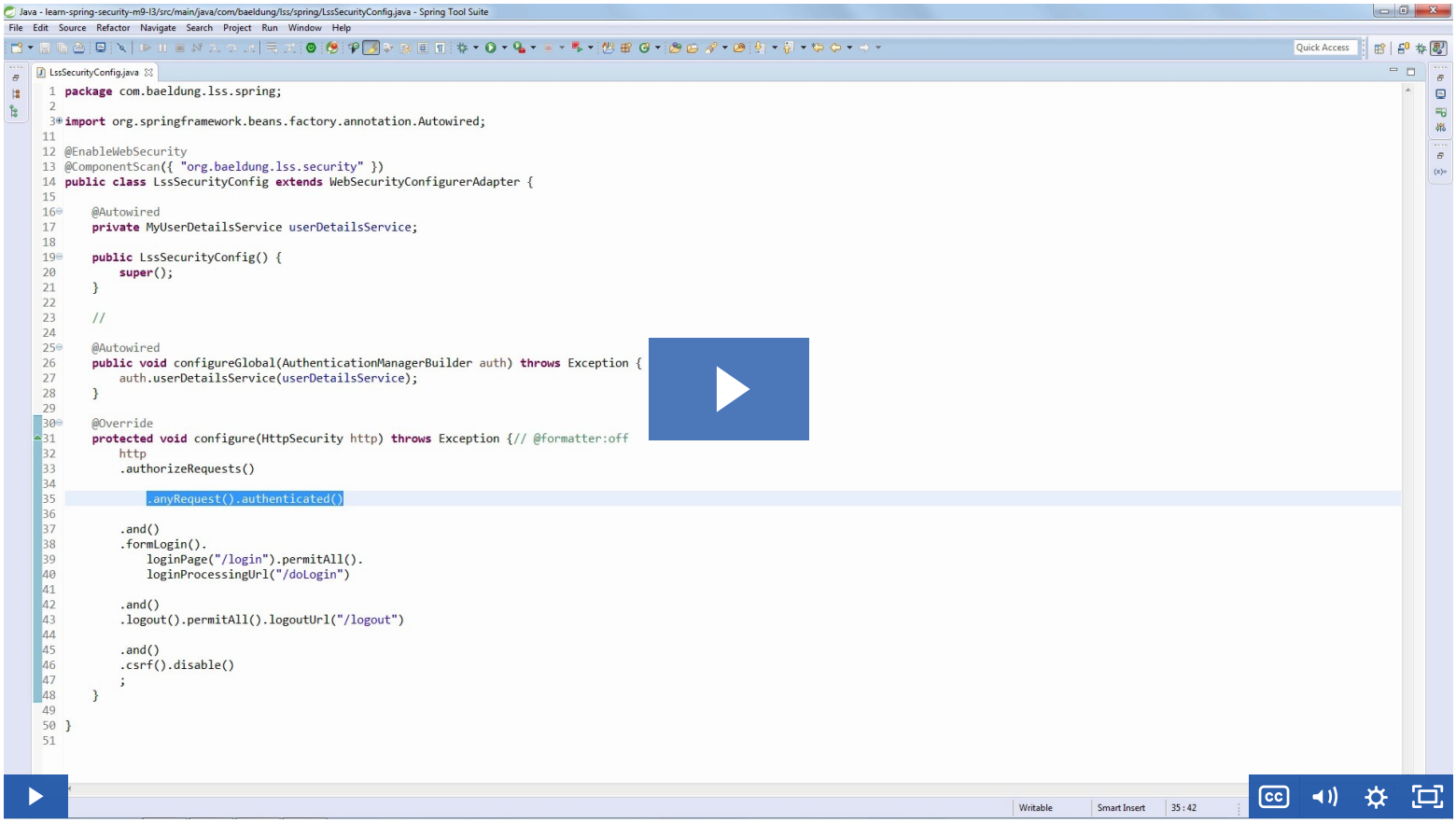
2. Lesson Notes

We're now going to actually implement the authorization structure discussed in the first part of this lesson.

3. Resources

- Spring Security – Roles and Privileges

Lesson 3: Secure Method Invocations with AOP



1. Goals

The goal of this lesson is to do method security in a centralized way, without using annotations.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m9-lesson3-start

If you want to skip and see the complete implementation, feel free to jump ahead and import: m9-lesson3

The credentials used in the code of this lesson are *test@test.com/pass* (*data.sql*)

Remember that in Lesson 1 - we had a look at two types of objects that were being secured. One was **the web request** - and that was secured with the standard filter, and the other was **a method invocation** - and that was secured with AOP.

Each has its own type of interceptor - the *FilterSecurityInterceptor* and the *MethodSecurityInterceptor* - both implementations of the *AbstractSecurityInterceptor* class.

Let's have a quick look at the *beforeInvocation* logic:

- get the "configuration attributes" associated with the present request
- try to authorize via the access decision manager
- if access is granted -> allow the invocation to proceed
- if access is not grated -> throw an *AccessDeniedException*

Note that these config attributes can be seen as simple Strings and it's the *AccessDecisionManager* implementation that has full control over what to do with them.

First, let's debug through the old logic and see what the default attributes are; we can see *authenticated* - as expected.

Now, before we secure a method invocation manually - let's discuss a few examples of when that's useful:

- if you don't have access to the source code and so cannot modify it (annotate it)
- or if you prefer to keep all security configurations centralized in one place

Now, let's set up these config attributes manually; we're going to use the specific method security configuration:

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
public static class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {
    //
}
```

Note that we can't use the main config class we have, because that one already extends a base class.

Now, we're going to provide **a simple implementation** for these attributes:

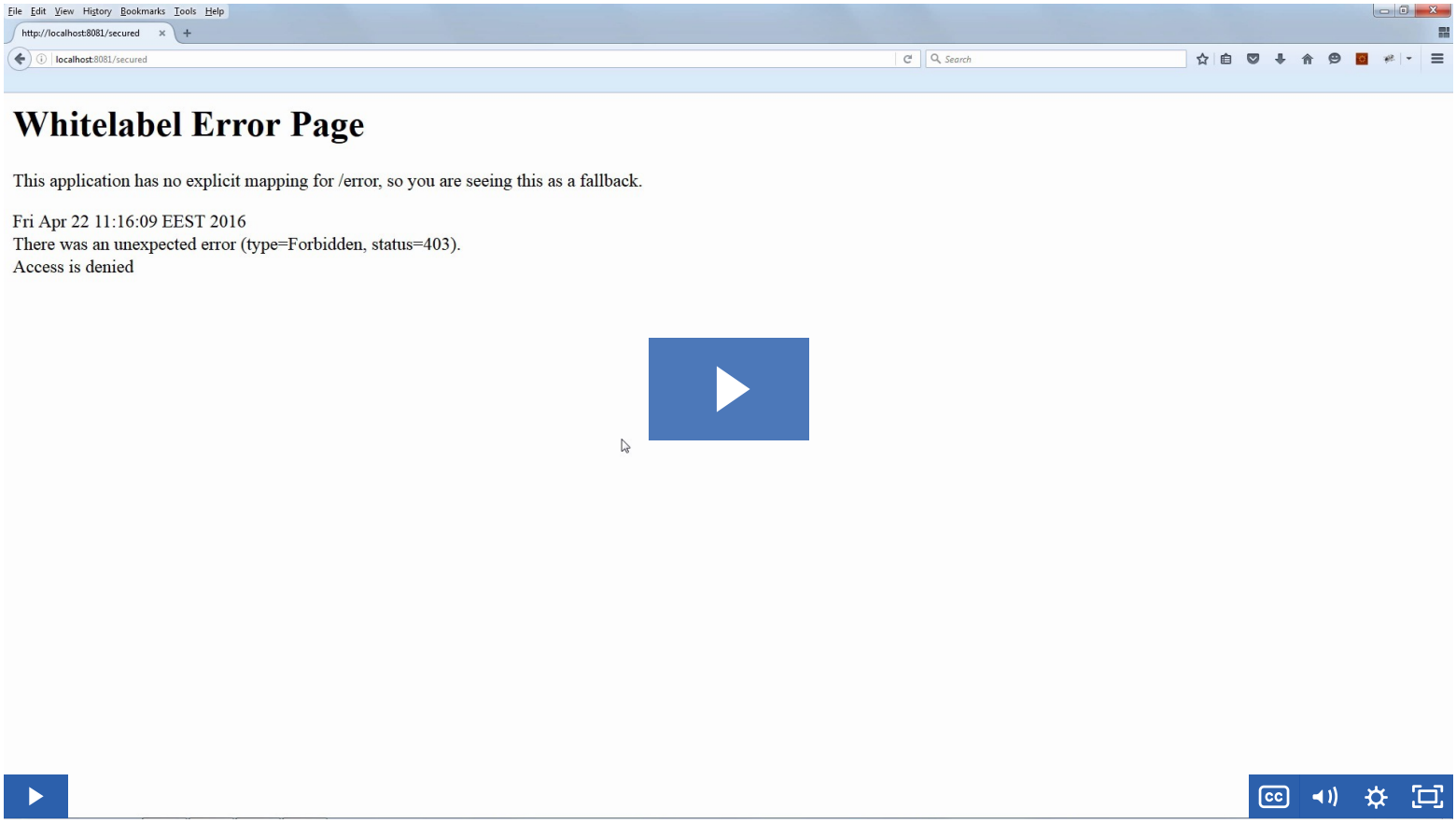
```
@Override
public MethodSecurityMetadataSource customMethodSecurityMetadataSource() {
    Map<String, List<ConfigAttribute>> methodMap = new HashMap<>();
    methodMap.put(
      "com.baeldung.lss.web.controller.UserController.createForm*",
      SecurityConfig.createList("ROLE_ADMIN"));
    return new MapBasedMethodSecurityMetadataSource(methodMap);
}
```

And we're done - now, the *createForm* method from the *UserController* will be secured with *ROLE_ADMIN* without utilizing any annotations on that method explicitly.

3. Resources

- AOP Alliance (MethodInvocation) Security Interceptor in the Official Reference

Lesson 4: A Custom AccessDecisionVoter



1. Goals

The goal of this lesson is to go beyond the standard voters and show how we can set up a custom one to implement an interesting scenario.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m9-lesson4

If you want to skip and see the complete implementation, feel free to jump ahead and import: m9-lesson5

The credentials used in the code of this lesson are:

user/pass (in-memory)

admin/pass (in-memory) (has the *ADMIN* role)

2.1. A Stricter Access Decision Manager

Let's start by changing the *AccessDecisionManager* implementation from *AffirmativeBased* to the more stricter *UnanimousBased*:

```
@Bean
public AccessDecisionManager unnanimous(){
    List<AccessDecisionVoter<? extends Object>> voters = Lists.newArrayList(
      new RoleVoter(), new AuthenticatedVoter(), new WebExpressionVoter());
    return new UnanimousBased(voters);
}
```

Now, let's wire it in:

```
.anyRequest().authenticated().accessDecisionManager(unnanimous())
```

And let's secure */secured* with an extra role:

```
.antMatchers("/secured").access("hasRole('ADMIN')")
```

We can now try to access */secured* and debug through the decision flow to see the stricter voting run.

2.2. The Custom Scenario

We're going to explore the following scenario - we need to be able to lock users out and have that lockout apply in real-time (not after a login).

The need for this kind of real-time lockout is simple - if a user is locked out, then they're a serious security concern so we need to make sure that their current session cannot be used to do any damage.

Before we start, not that there are multiple ways to implement this kind of scenario - this is just one of them.

Let's first create the new voter:

```
public class RealTimeLockVoter implements AccessDecisionVoter<Object> { ...
```

Now, before implementing the actual voting logic, let's create the very simplistic cache for users that are locked out:

```
public final class LockedUsers {
    private static final Set<String> lockedUsersSets = Sets.newHashSet();
    private LockedUsers() {}
    public static final boolean isLocked(final String username) {
        return lockedUsersSets.contains(username);
    }
    public static final void lock(final String username) {
        lockedUsersSets.add(username);
    }
}
```

Finally - let’s get back to the voting logic:

```
if (LockedUsers.isLocked(authentication.getName())) {
    return ACCESS_DENIED;
}
return ACCESS_GRANTED;
```

And we’re done - simple and to the point.

Now if we go through a simple scenario

- login with *admin* (so that we can access */secured*)

- then (with the help of the Display view in Eclipse) we add the user to the locked cache

- and refresh => **we're locked out**