Intro to the Certification Class

General Information

Welcome to the new Certification Class in Learn Spring Security.

Given this class includes the Master Class, let's start with the structure.

The Master Class contains the following 5 modules:

- ACL with Spring Security (3 lessons)
- Advanced REST API Security (3 lessons)
- OAuth Beyond the REST API (3 lessons)
- Two-Factor Authentication (2 lessons)
- Advanced Spring Security Scenarios (4 lessons)

Each Module is organized in several video Lessons - containing lesson notes and other resources.

The material is meant to be experienced as video, and **the lesson notes should be used as a reference not as comprehensive documentation**.

Finally, have a look at the Course Intro page for additional, course-wide details.

Resources

The git repository of the project, hosted over on Github:

- The Module 11 branch

- The Module 12 branch

- The Module 13 branch

- The Module 14 branch

- The Module 15 branch

Lesson 1: Introduction ACL and Domain Object Security

1. Goal

The goal of this lesson is to introduce Domain Object Security and Access Control Lists (ACL).

2. Lesson Notes

As you're using the git repository to get the project - you should be on the module11 branch.

Or, on the corresponding Spring Boot 2 based branch: module11-spring-boot-2.

This lesson is theoretical and doesn't have/need accompanying code.

2.1. The Need for Granular Access Control

First, let’s see why we even need this kind of fine-grained type of authorization control in the first place.

Let’s look at a scenario.

The users in our system control some objects - let’s call them possessions. By default, **each user has the right to work only with his own possession objects**.

But, let’s say that we’re **introducing the concept of borrowing**. A user can now borrow a possession from another user.

More technically, the owner will grant some privileges to the borrower over that particular possession.

The previous authorization model we’ve been using is **simply not flexible enough to handle this kind of scenario**.

Remember that, in the traditional authorization model we discussed up until this point, you define authorities per type of object. So you can for example define that users that have this authority can do this action - on ALL objects of this type. What you can't define is - users that have this authority can do this action on object A of this type, but not on object B of the SAME type.

That’s why we need a new, more flexible and more granular mode.

2.2. Potential Solutions

Now, this problem can be solved manually, by combining per-type authorization and then custom business logic doing extra checks.

What's the problem with that solution though?

Typically - this becomes complex to manage, spreads the responsibility of authorization into multiple parts of the system and is generally not maintainable.

More importantly, doing this manually will mostly re-invent the wheel - since there’s already a solid, mature solution specifically for this kind of problem.

The new model will **allow us to define the security semantics of a specific domain object**, not just a class/type of objects.

In short, it’s a generic way of defining authorization semantics for each domain entity using what is called an access control list - ACL. It's going to allow us full granular control over exactly which users in the system can access exactly which objects.

2.3. ACL

So let’s now take a quick look at the main concepts in ACL.

We’ll start with the database structure and see exactly what the framework needs.

First, let’s see what the structure of this ACL data looks like:

- the Security Identity - the SID
- the Domain Object
- the ACL Entry

Generally, **the SID** will represent the principal that gets access to the domain object. What's interesting though and adds even more flexibility into this model is that the SID can also represent an authority

The Domain Object is composed out of two entities:

- Class - the actual Java class of the entity
- Object Identity - the main identifier of the entity we are trying to secure

Finally, we have **the ACL Entry -** this represents the actual permissions that the principal has on the domain objects.

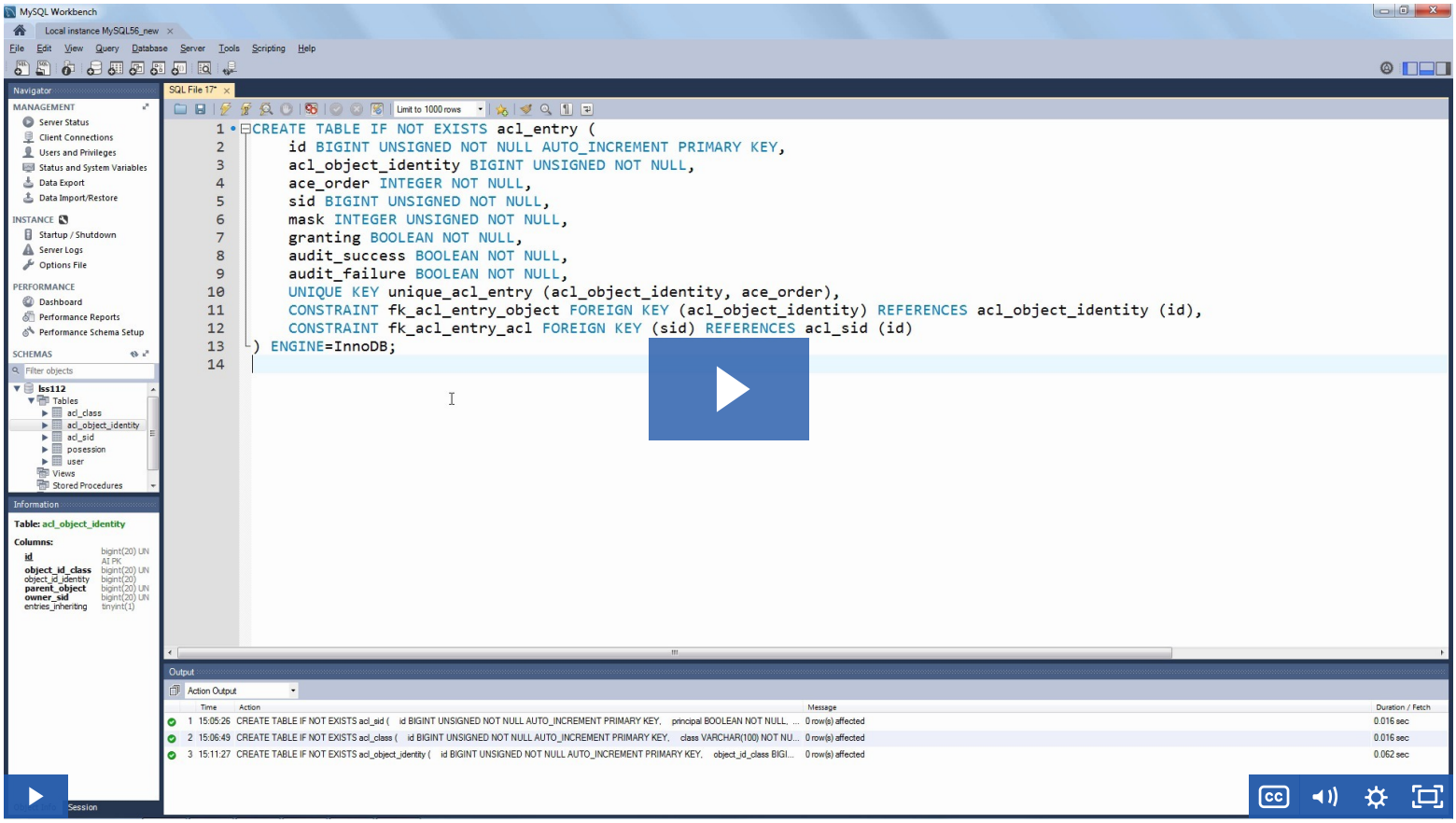
By default these are: *read, write, create, delete, admin* - and they're represented with an integer bit mask - so we have 32 bits available (and 5 used).

So, because there are 32 available bits and we're only using 5, we can add new, custom types of access if we need to

3. Resources

- ACL in the Spring Security Reference

Lesson 2: The Data Structure of ACL



1. Goals

The focus of this lesson is on developing the data structure that we're going to need to do ACL with Spring Security.

2. Lesson Notes

This lesson is theoretical and doesn't have/need accompanying code.

First, it's important to understand that the Spring Security implementation is generic enough that you can develop non-SQL implementations as well (and we will do that).

But **the out of the box implementation is entirely focused on SQL** persistence for the ACL artifacts.

With that in mind, the very first thing we'll do is - we'll create the DB structure for our basic domain. We'll then create the DB structure for ACL, and we'll wrap up by actually creating some test data in the system.

And we're going to use MySQL in this lesson, just to be able to show everything clearly with the MySQL Workbench.

2.1. The Domain

Before we get to ACL, let's start with our domain. Of course we're going to continue using the domain we already have - the users.

But also remember we mentioned in the previous lesson that we're going to be working with **a new entity - a *Possession***:

```
@Entity
public class Possession {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @ManyToOne
    @JoinColumn(name = "owner_id", nullable = false)
    private User owner;
    // standard getters and setters
}
```

Let's now run the system, check out our existing DB structure, and move on.

2 simple tables - the *user* table and a new *possesion* table will be generated.

2.2. The ACL SQL Structure

Now, let's jump right into ACL and let's start creating the structure.

One quick note here is that the Spring Security reference has scripts for other DBs, but since we're using MySQL here, we have slightly modified the scripts to adhere to the MySQL syntax.

We're going to first set up the SID table:

```
CREATE TABLE IF NOT EXISTS acl_sid (
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    principal BOOLEAN NOT NULL,
    sid VARCHAR(100) NOT NULL,
    UNIQUE KEY unique_acl_sid (sid, principal)
) ENGINE=InnoDB;
```

We're then going to set up the class and object identity tables:

```
CREATE TABLE IF NOT EXISTS acl_class (
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    class VARCHAR(100) NOT NULL,
    UNIQUE KEY uk_acl_class (class)
) ENGINE=InnoDB;
```

```
--
CREATE TABLE IF NOT EXISTS acl_object_identity (
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    object_id_class BIGINT UNSIGNED NOT NULL,
    object_id_identity BIGINT NOT NULL,
    parent_object BIGINT UNSIGNED,
    owner_sid BIGINT UNSIGNED,
    entries_inheriting BOOLEAN NOT NULL,
    UNIQUE KEY uk_acl_object_identity (object_id_class, object_id_identity),
    CONSTRAINT fk_acl_object_identity_parent FOREIGN KEY (parent_object) REFERENCES acl_object_identity (id),
    CONSTRAINT fk_acl_object_identity_class FOREIGN KEY (object_id_class) REFERENCES acl_class (id),
    CONSTRAINT fk_acl_object_identity_owner FOREIGN KEY (owner_sid) REFERENCES acl_sid (id)
) ENGINE=InnoDB;
```

Finally, let's create the actual ACL entry table:

```
CREATE TABLE IF NOT EXISTS acl_entry (
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    acl_object_identity BIGINT UNSIGNED NOT NULL,
    ace_order INTEGER NOT NULL,
    sid BIGINT UNSIGNED NOT NULL,
    mask INTEGER UNSIGNED NOT NULL,
    granting BOOLEAN NOT NULL,
    audit_success BOOLEAN NOT NULL,
    audit_failure BOOLEAN NOT NULL,
    UNIQUE KEY unique_acl_entry (acl_object_identity, ace_order),
    CONSTRAINT fk_acl_entry_object FOREIGN KEY (acl_object_identity) REFERENCES acl_object_identity (id),
    CONSTRAINT fk_acl_entry_acl FOREIGN KEY (sid) REFERENCES acl_sid (id)
) ENGINE=InnoDB;
```

2.3. The Setup Data

Finally, let's create some basic example data here to make sure that our system is actually populated.

We're going to start with the data for our two main domain entities - *User* and *Possession*:

```
INSERT INTO User (id, email, password)
VALUES
(1, 'eugen@email.com', 'pass'),
(2, 'eric@email.com', '123');
--
INSERT INTO Possession (id, name, owner_id)
VALUES
(1, 'Eugen Possession', 1),
(2, 'Common Possession', 1),
(3, 'Eric Possession', 2);
```

Then we'll move to the ACL artifacts for these:

```
INSERT INTO acl_sid (id, principal, sid)
VALUES
(1, 1, 'eugen@email.com'),
(2, 1, 'eric@email.com');
--
INSERT INTO acl_class (id, class)
VALUES
(1, 'com.baeldung.lss.model.Possession');
--
INSERT INTO acl_object_identity
(id, object_id_class, object_id_identity, parent_object, owner_sid, entries_inheriting)
VALUES
(1, 1, 1, NULL, 1, 1), -- Eugen Possession object identity
(2, 1, 2, NULL, 1, 1), -- Common Possession object identity
(3, 1, 3, NULL, 1, 1); -- Eric Possession object identity
```

Errata: Note that the video shows an incorrect value for the data in the *acl_sid* table - make sure you use the values in the lesson notes here. Thanks.

Finally, the very last part of the ACL artifacts - the actual ACL entries that decide the kind of access each user has for the possessions.

Now, as you can already see from the name of these users and possessions, we intend to make sure that each has access to their own possession.

And that the common possession is actually shared and both users can access it:

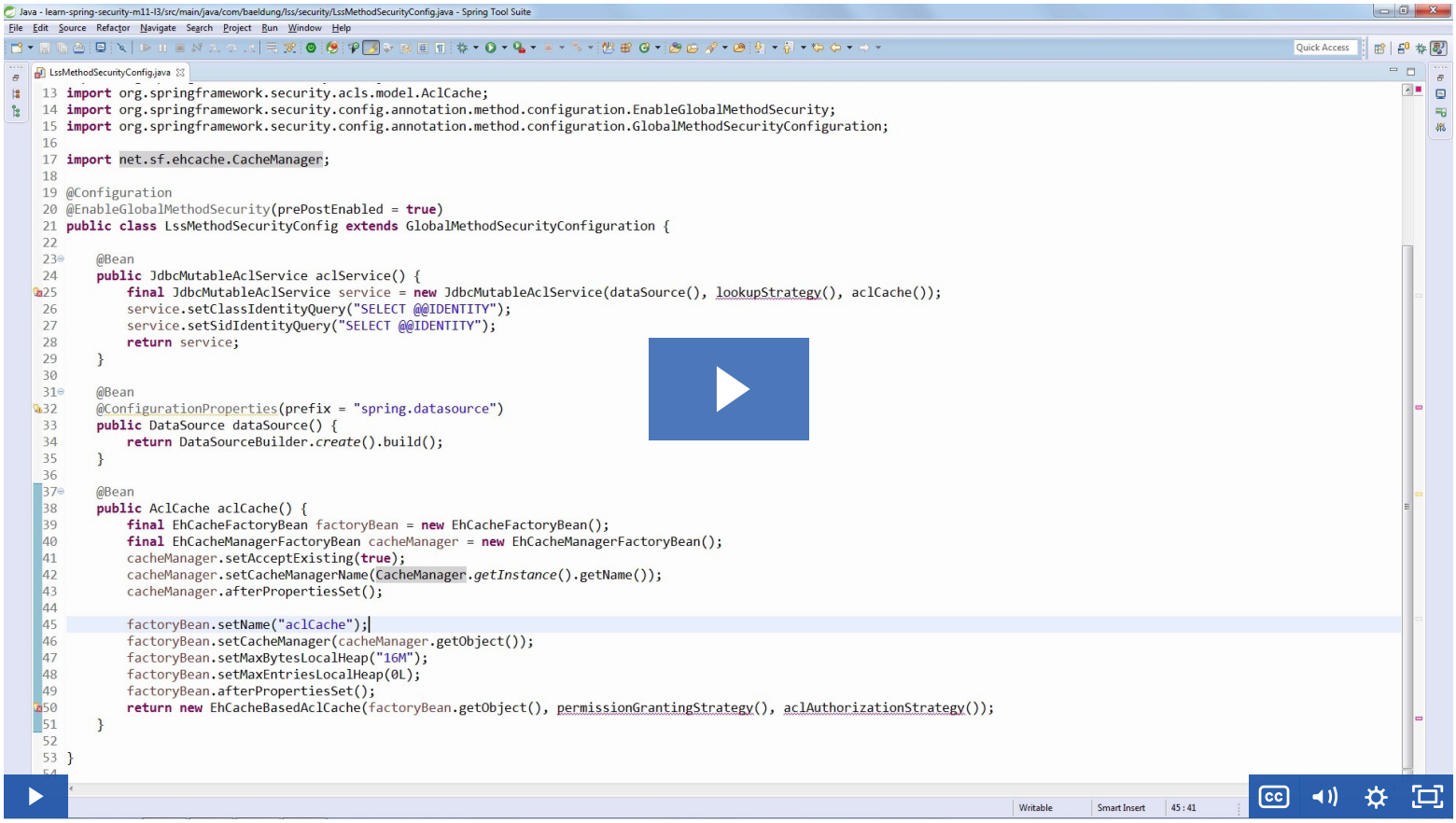
```
INSERT INTO acl_entry
(id, acl_object_identity, ace_order, sid, mask, granting, audit_success, audit_failure)
VALUES
(1, 1, 0, 1, 16, 1, 0, 0), -- eugen@email.com has Admin permission for Possession 1
(2, 2, 0, 1, 16, 1, 0, 0), -- eugen@email.com has Admin permission for Common Possession 2
(3, 2, 1, 2, 1, 1, 0, 0),  -- eric@email.com has Read permission for Common Possession 2
(4, 3, 0, 2, 16, 1, 0, 0); -- eric@email.com has Admin permission for Eric Possession 3
```

And that's it - we created the ACL structure as well as example data to use that structure.

3. Resources

- The ACL Schema in the Reference

Lesson 3: ACL with Spring Security - part 1



1. Goals

The simple goal of this lesson is to do a full ACL implementation with Spring Security.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m11-lesson3

If you want to skip and see the complete implementation, feel free to jump ahead and import: m11-lesson4

Before we start, let's step back and look at ACL in Spring Security. When we do an ACL implementation, that means we get access to this new and very flexible way of doing authorization.

However, it's important to understand that we're not limited to it - and **we can of course still use all of the earlier ways of doing authorization**, such as checking roles or authorities.

In the previous lesson, we had a look at the DB structure that is the backbone of ACL.

Now, we're on to the core - the actual implementation with Spring Security.

First, we'll add the dependency:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-acl</artifactId>
</dependency>
```

We'll also need some caching support later on, so let's define this now as well:

```
<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache-core</artifactId>
    <version>2.6.11</version>
</dependency>
```

Now, the first thing we'll need is to configure the method security support - because we're going to use that when configuring our authorization with ACL.

So, just as we've done in previous lessons, we'll define a separate method security config class:

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

Now, we're going to define a new permission evaluator, specifically for ACL. However, before we get there, we first need to build out the infrastructure - the beans - that are going to support ACL.

First let's define the ACL service - and we'll go with the official, JDBC backed implementation that the framework provides:

```
@Bean
public JdbcMutableAclService aclService() {
    JdbcMutableAclService service =
      new JdbcMutableAclService(dataSource(), lookupStrategy(), aclCache());
    service.setClassIdentityQuery("SELECT @@IDENTITY");
    service.setSidIdentityQuery("SELECT @@IDENTITY");
    return service;
}
```

One quick note is that these 2 simple SQL queries are MySQL specific, so if you're supporting multiple databases, you might want to put these in configuration, not in code.

Now we need the data source, a lookup strategy and a cache. This is why we defined the ehcache dependency earlier.
So let's define these 3 new beans:

```
@Bean
public LookupStrategy lookupStrategy() {
    return new BasicLookupStrategy(
      dataSource(),
```

```
      aclCache(),
      aclAuthorizationStrategy(),
      permissionGrantingStrategy());
}
@Bean
public AclCache aclCache() {
    EhCacheFactoryBean factoryBean = new EhCacheFactoryBean();
    EhCacheManagerFactoryBean cacheManager = new EhCacheManagerFactoryBean();
    cacheManager.setAcceptExisting(true);
    cacheManager.setCacheManagerName(CacheManager.getInstance().getName());
    cacheManager.afterPropertiesSet();
    factoryBean.setName("aclCache");
    factoryBean.setCacheManager(cacheManager.getObject());
    factoryBean.setMaxBytesLocalHeap("16M");
    factoryBean.setMaxEntriesLocalHeap(0L);
    factoryBean.afterPropertiesSet();
    return new EhCacheBasedAclCache(
      factoryBean.getObject(),
      permissionGrantingStrategy(),
      aclAuthorizationStrategy())
}
@Bean
@ConfigurationProperties(prefix = "spring.datasource")
public DataSource dataSource() {
    return DataSourceBuilder.create().build();
}
```

The lookup strategy and the data source is one lines and super simple to understand.

The cache has a bit more configuration - which we can safely skip that for now.

But notice that **we still need a couple more beans**. Let's define the two remaining beans:

- a permission granting strategy - this allows us to actually customize the decision to grant a permission (or not) based on the ACL entry
- an authorization strategy - which deals with access administrative methods

OK, so now, with all of this in place, we're finally at the point where we can create the main bean that we needed to create in the first place - **the permission evaluator**:

```
@Bean
public AclPermissionEvaluator aclPermissionEvaluator() {
    AclPermissionEvaluator aclPermissionEvaluator =
      new AclPermissionEvaluator(aclService());
    return aclPermissionEvaluator;
}
```

Now that everything compiles and we can finally wire this new bean into the method security config we have here:

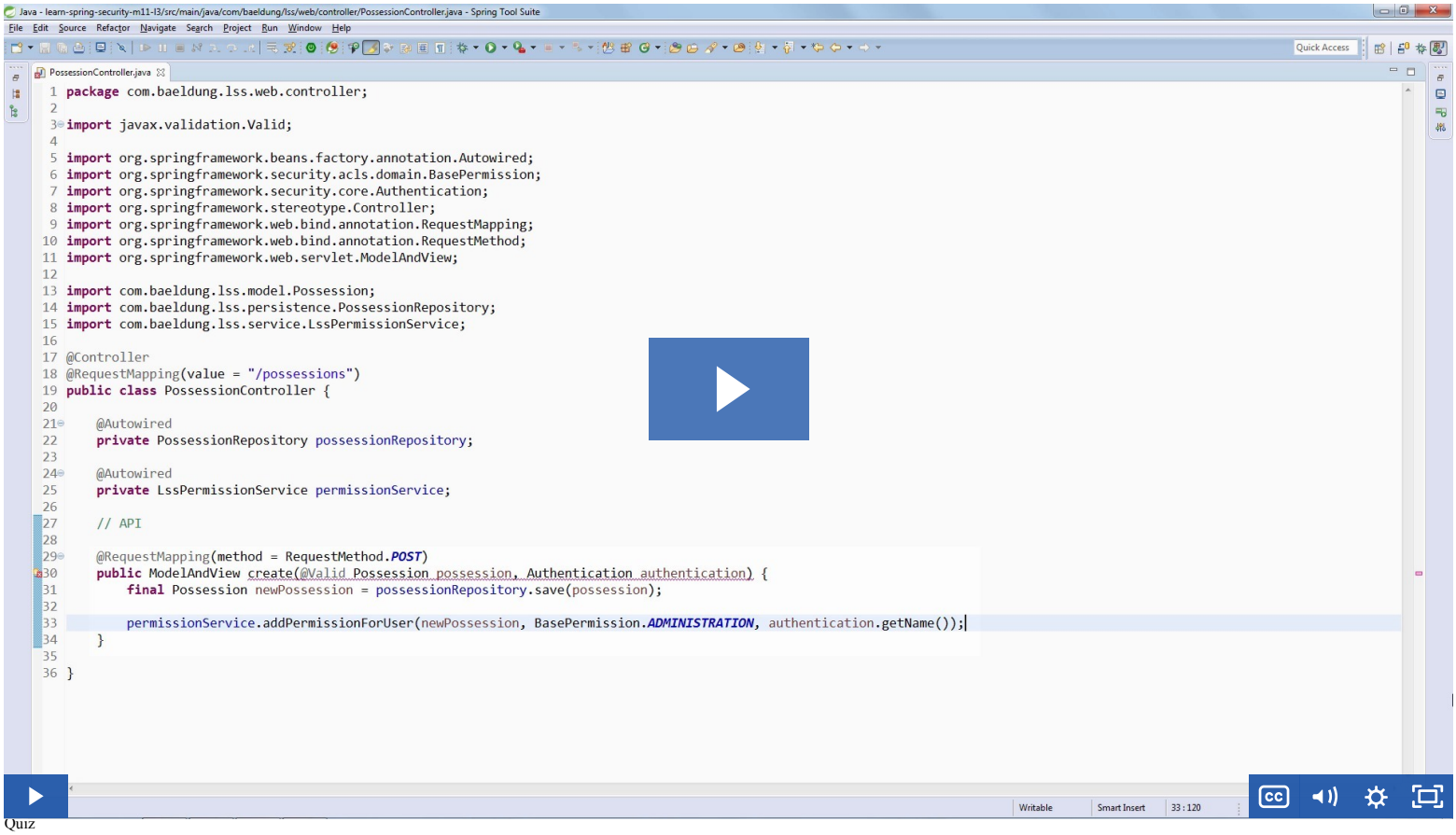
```
@Override
protected MethodSecurityExpressionHandler createExpressionHandler() {
    DefaultMethodSecurityExpressionHandler expressionHandler =
      new DefaultMethodSecurityExpressionHandler();
    expressionHandler.setPermissionEvaluator(aclPermissionEvaluator());
    return expressionHandler;
}
```

And that's it - this is the full configuration of ACL with Spring Security.

Finally, a quick note about the *Possession* entity - when you create it, make sure it implements the *IEntity* marker interface:

```
@Entity
public class Possession implements IEntity {
    ...
}
```

Lesson 3: ACL with Spring Security - part 2



Quiz

1 / 2

On a high level, how is ACL implemented in Spring?

It’s implemented using URL level security configurations with the help of an AclPermissionEvaluator

It uses Spring’s method level security with an AclPermissionEvaluator based security handler

It’s implemented using both URL level and method level security features

‹ Back | Check | Continue ›