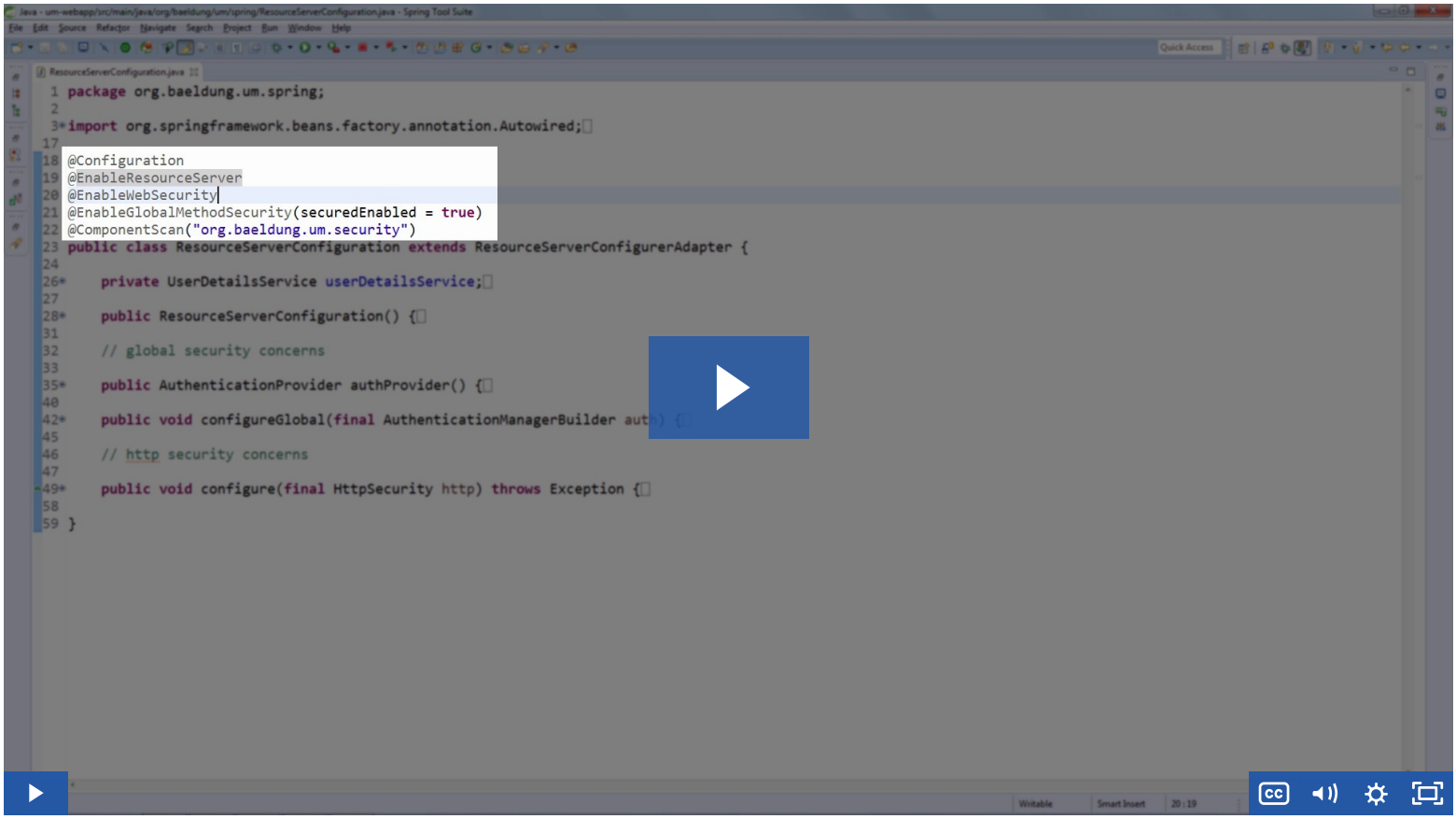
Lesson 1: Setup OAuth2 with Spring Security



1. Goals

The simple goal of this lesson is to implement a fully working OAuth2 flow with Spring Security.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m12-lesson1

If you want to skip and see the complete implementation, feel free to jump ahead and import: m12-lesson2

2.1. Client Credentials and User Credentials

Let's start with a quick note to clarify the different types of credentials in OAuth2.

First, we have the Client - which generally has **client credentials**.

The Client can of course be used by multiple users.

And then we have users - witch have their own **user credentials**.

It's a very simple distinction, but one that's worth making here, at the very beginning.

2.2. The Authorization Server

We're defining the Authorization Server configuration here, with the *@EnableAuthorizationServer* annotation and the standard configurer-adapter style of configuration.

We're going to go with the Password Flow because the first usecase for OAuth is going to be the suite of live tests - which is a trusted type of client.

A quick note - **the Authorization Server endpoint is itself secured** via Basic Authentication with the client credentials - the client id and the secret.

The Tokens endpoint for a local deployment: *http://localhost:8082/um-webapp/oauth/token*

2.3. The Resource Server

The Resource Server is effectively the API - so, the config for it will be our main security configuration.

A quick side-note here is that this configuration is almost exactly the same with our previous config. The only few differences to get a full OAuth config are - we're no longer defining Basic Auth and we're using the *@EnableResourceServer* annotation and the corresponding configurer-adapter.

2.4. The “password” Flow

We’re giving the master key (the password) to the client application. This flow will skip the */authorization* endpoint all together and directly talk to the */token* endpoint to generate an access token.

Here's a quick example interaction with the /token endpoint:
POST
http://localhost:8082/um-webapp/oauth/token?grant_...
live-test:bGl2ZS10ZXN0

And a potential response from the server:
{
"access_token": "748493f7-c17c-44c7-b4e6-e9c95dcfa511",
"token_type": "bearer",
"expires_in": 3599,

"scope": "um-webapp"
}

2.5. The Live Tests and rest-assured

The flow from the POV of the client:
- first - we retrieve the access token
- then, we set that up very simply with rest-assured

Note that we're interacting with the */token* endpoint on every request and not re-using the token, so this is a potential improvement in the way we're handling the flow on the client side.

Finally, the rest-assured driven test that retrieves an Access Token and then uses it to consume the API is: *AuthorizationLiveTest*.

2.6. Spring Security OAuth2 Config - High Level

Let's have a quick, high-level look at the typical Spring Security OAuth2 configuration:

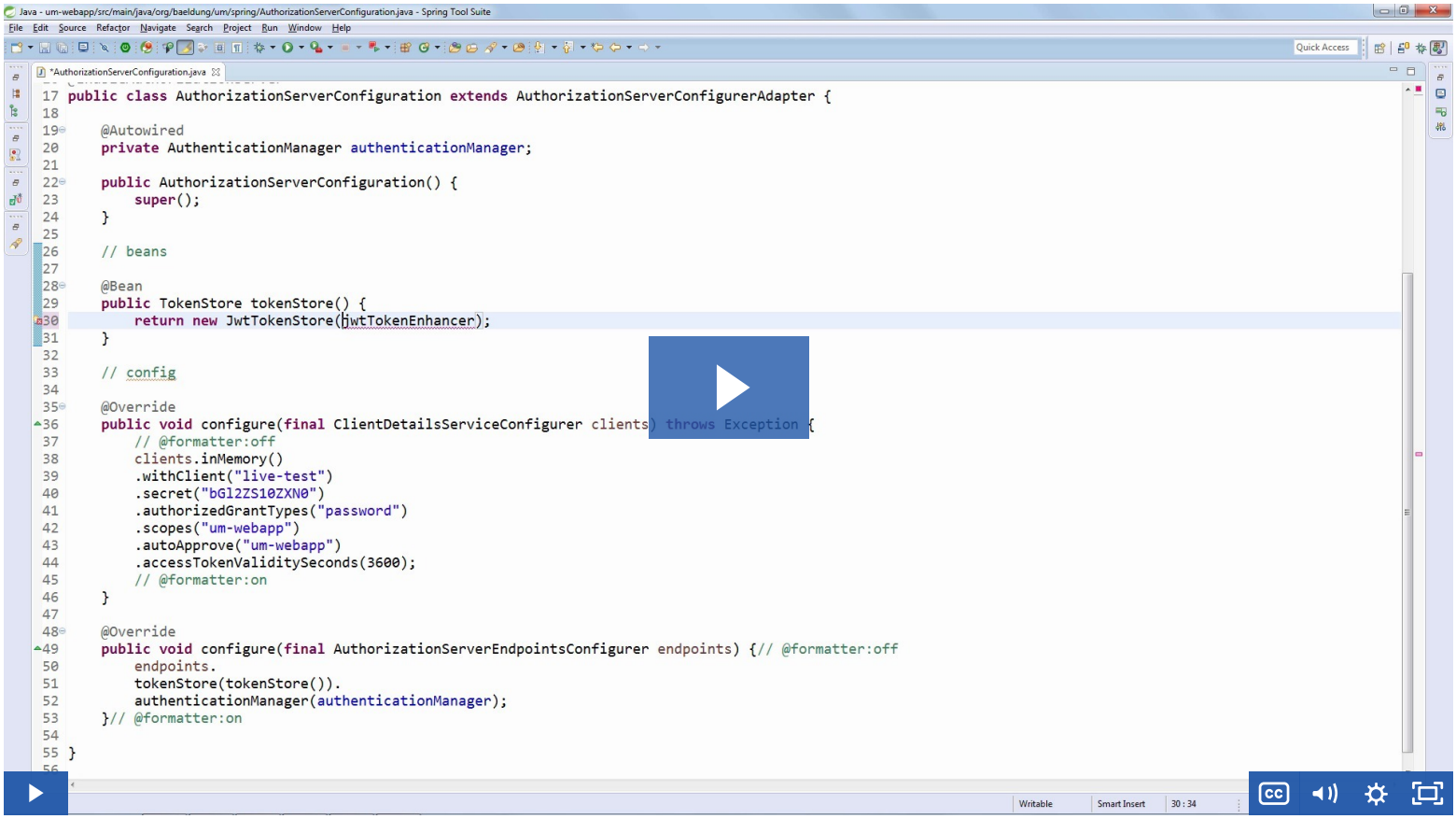
- the token services - has the responsibility of generating the Access and Refresh tokens and storing them; by default, it's able to create UUID-based tokens
- the token enhancer - different enhancers can generate different types of tokens; the token services delegates to it to generate the token (ex: JwtAccessTokenConverter)
- the token converter - this converts between the encoded value and the normal value of the token; it's also able to extract auth information out of the token

3. Resources

- Spring Security OAuth

- OAuth 2 Developers Guide

Lesson 2: Tokens, OAuth2 and JWT



1. Goals

The goal of this lesson is to understand what tokens are, why JWT is a solid option and how to set it up with Spring Security.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m12-lesson2

If you want to skip and see the complete implementation, feel free to jump ahead and import: m12-lesson3

2.1. Token Implementations

SAML (or the WS* space)
- XML based

- many encryption and signing options

- expressive but you need a pretty advanced XML stack

Simple Web Token

- joint venture between Microsoft, Google, Yahoo

- created as a direct reaction to making a much simpler version of SAML

- to simple, not enough cryptographic options (just symetric)

JWT (JSON Web Tokens)

- the idea is that you are representing the token using JSON (widely supported)

- symmetric and asymmetric signatures and encryption

- less options/flexibility than SAML but more than SWT

- JWT hit the sweet spot and became widely adopted pretty quickly

- JWT - an emerging protocol (very close to standardization)

2.2. JWT Token Structure

First, let's understand the very high level structure of a JWT Token:

- Header (Base64Url encoded)
- Payload (Base64Url encoded)
- Signature

The Header
- the type of token (JWT)

- the hashing algorithm being used

The Payload
- consists of the claims made by the token:

- reserved claims = predefined claims – not mandatory, but recommended
- public claims = extensions, but still regulated by IANA (the Internet Assigned Numbers Authority)
- private claims = the fully custom claims created by each implementation

The Signature

- the signature is made up of:

- the encoded header
- the encoded payload
- the signature

And of course signed with the algorithm specified in the header.

2.3. JWT with Spring Security OAuth

For the Authorization Server:

- we're defining the *JwtAccessTokenConverter* bean and the *JwtTokenStore*

- we're also configuring the endpoint to use the new converter

Note that we're using symmetric signing - with a shared signing key.

For the Resource Server:

- we should define the converter here as well, using the same signing key

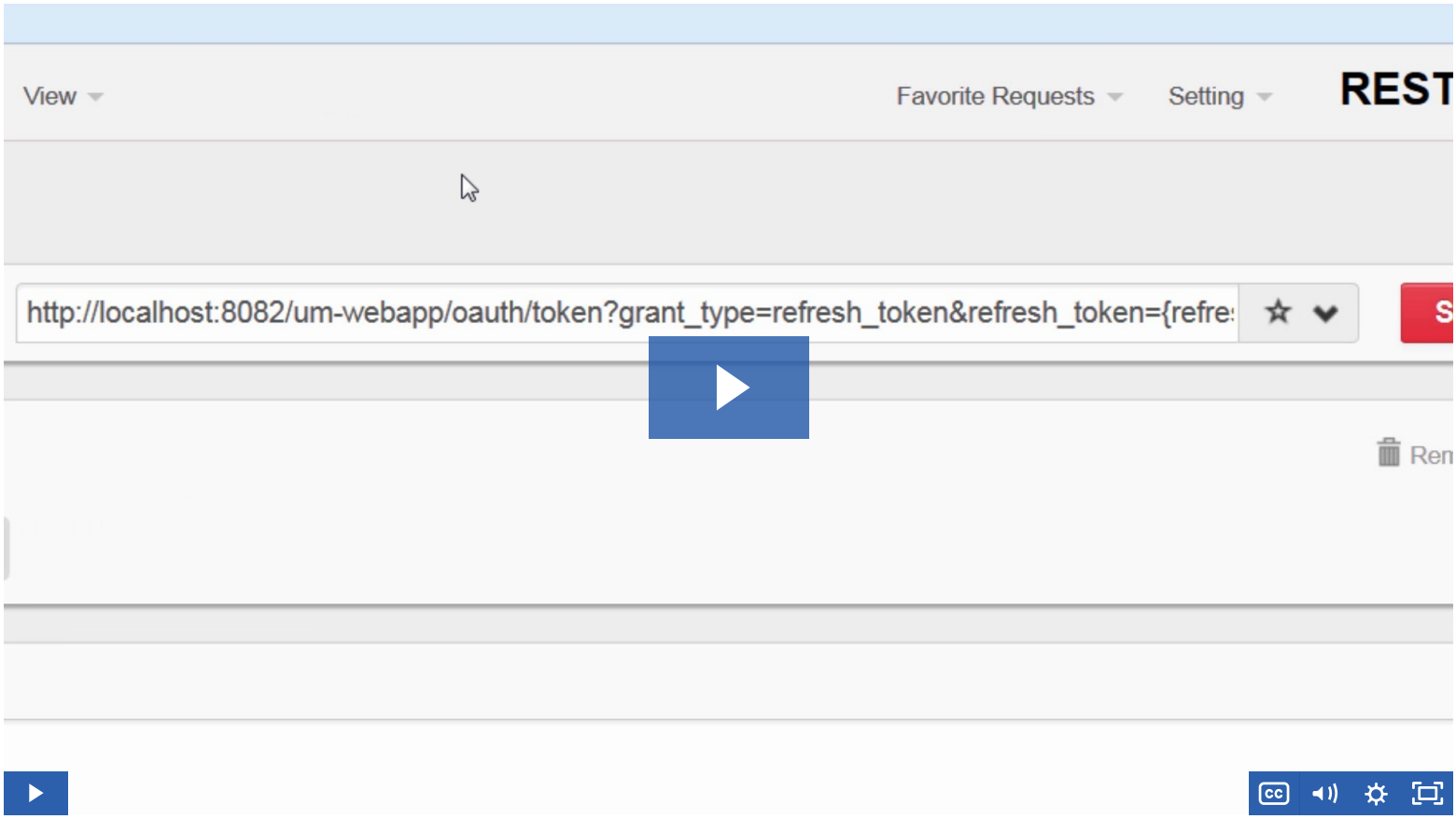
Note that we don't have to because we're actually sharing the same Spring context in this case. If the Authorization Server would have been a separate app - then we would have needed this converter, configured exactly the same as in the Resource Server.

3. Resources

- jwt.io

- Spring Security and Angular JS

Lesson 3: Refreshing a Token



1. Goals

The focus and goal of this lesson is to show you how to obtain a new Access Token using the Refresh Token.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m12-lesson3

If you want to skip and see the complete implementation, feel free to jump ahead and import: m12-lesson4

One quick note before starting the lesson - you'll notice that the video is showing two clients in the config - *live-test* and *um*. You can ignore the second client - we're not using it in the lesson.

2.1. Check the Access Token

After enabling the check functionality in the Spring Security configuration:

*GET http://localhost:8082/um-webapp/oauth/check_token?token=**{access_token}***

Note that, depending on what kind of access you configured for the check endpoint, you may still need the standard Basic Authentication on this request.

2.2. Refresh the Access Token

After enabling the refresh token functionality in the Spring Security configuration, and after retrieving the first Access Token (and Refresh Token), we can now:

*POST http://localhost:8082/um-webapp/oauth/token?grant_type=refresh_token&refresh_token=**{refresh_token}***

We're getting back both the new Access Token that we were expecting, but also a new Refresh Token. Note that both the old and this new Refresh Tokens will be valid in the system (until they expire).

3. Resources

- About Refresh Tokens on auth0