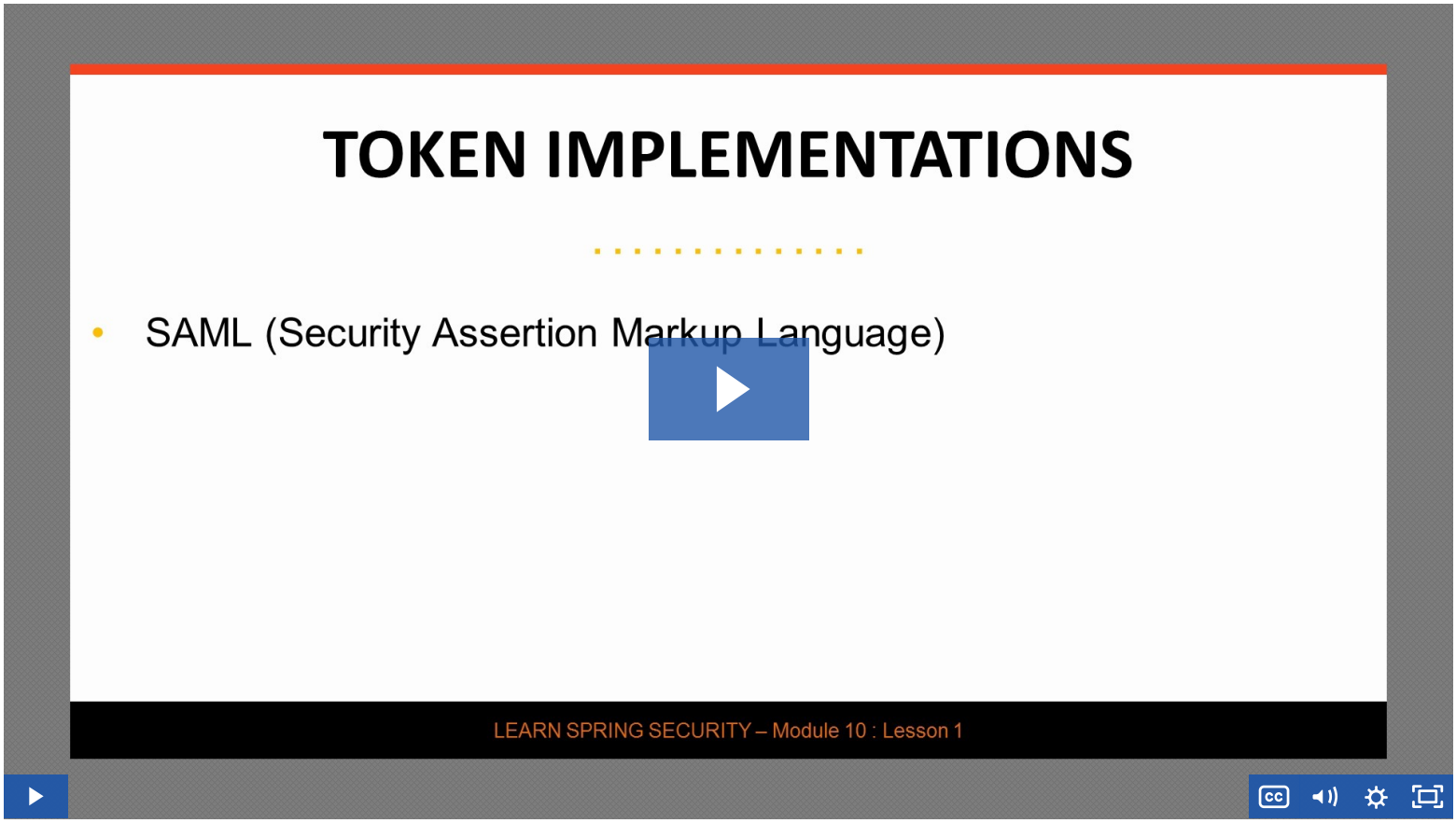
Lesson 1: The Basics of API Security



1. Goal

The goal of this first lesson is to give you a quick intro to the potential options for securing a REST API.

2. Lesson Notes

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

2.1. API Security Options

The API Security ecosystem has quite a number of options:

- Basic Authentication (stateless)
- Digest Authentication (stateless)
- Form-based Authentication (stateful)
- OAuth 2 (stateful)
- OAuth 2 + JWT (stateless)
- Custom Token Implementation (stateful or stateless)

There are a few additional alternatives - such as X.509 Authentication - but these only make sense in very specific scenarios.

2.2. 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)

2.3. JWT structure

A JWT token has 3 parts:

The Header

The header carries 2 parts:

- declaring the type, which is JWT
- the hashing algorithm to use (HMAC SHA256 in this case)

Here's an header example:

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

The Payload

The payload contains the JWT Claims.

Simply put, a claim is a piece of information we want to transmit with our token.

There are multiple claims that we can provide - this includes registered claims, public claims, and private claims.

Here's a quick example payload:

```
{
    "iss": "scotch.io",
    "exp": 1300718380,
    "name": "Eugen",
     "admin": true
}
```

The Signature

Finally, the third and final part of our JSON Web Token is going to be the signature.

This signature is made up of a hash of the following components:

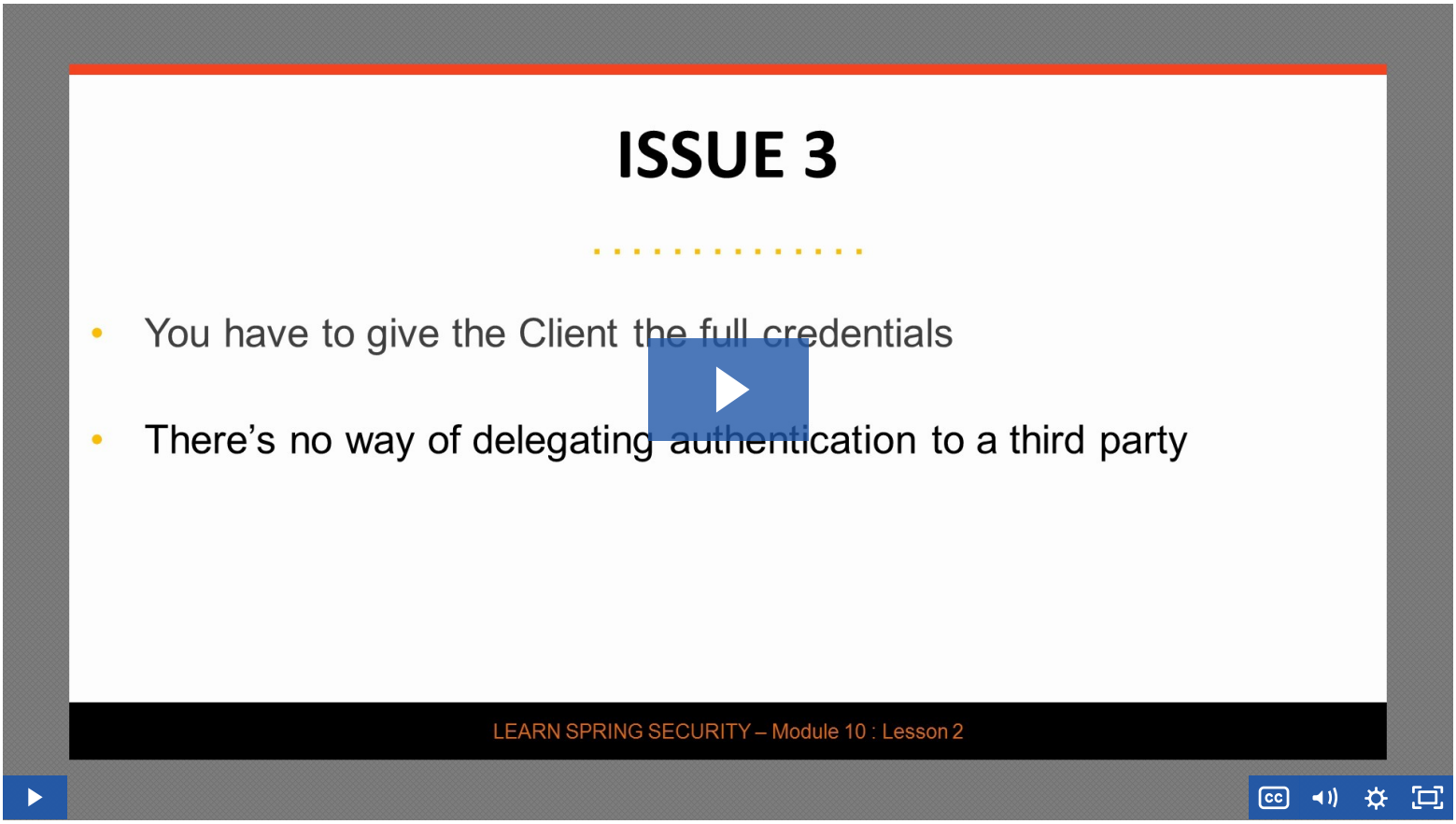
- the header
- the payload
- secret

This is the way that our server will be able to verify existing tokens and sign new ones.

3. Resources

- jwt.io

Lesson 2: Basic Authentication for the API



1. Goal

The simple goal of this lesson is to introduce the Basic Authentication mechanism as a potential solution to secure the API, and then discuss its limitations and the scenario where it doesn't make sense.

2. Lesson Notes

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

2.1. The Basics

Basic Authentication is the lowest common denominator for securing an API.

Everybody knows how to deal with it - it's completely ubiquitous on the client side and in most networking devices as well.

It also has very good support in Spring Security and Spring Boot.

Simply put, it uses the *Authorization* header to transmit Base64 encoded credentials over the wire.

Here's a quick example:

```
curl https://$username:$password@thehost:port/resource
```

2.2. The Problems

So, what are the issues with this solution?

First, Base64 encoded credentials can be easily converted to plaintext. This means that the entire API needs to run over HTTPS.

And even over HTTPS, this only solves part of the problem. SSL only protects the data over the wire; once the data hits the server, there may still be internal routing, logging, etc that potentially expose credentials.

Another issue is the password is sent repeatedly, for each request (slightly larger attack window).

You also have to give the Client the full master key - the actual credentials, so if these get compromise, the impact will be significantly greater than with a token based solution.

Another issue is that the password is cached by the browser and is automatically sent to the server on new requests.

This very obviously opens up the system to potential CSRF vulnerabilities without extra protection.

Next - there's no way to use Two-factor authentication, as there's no mechanism for that.

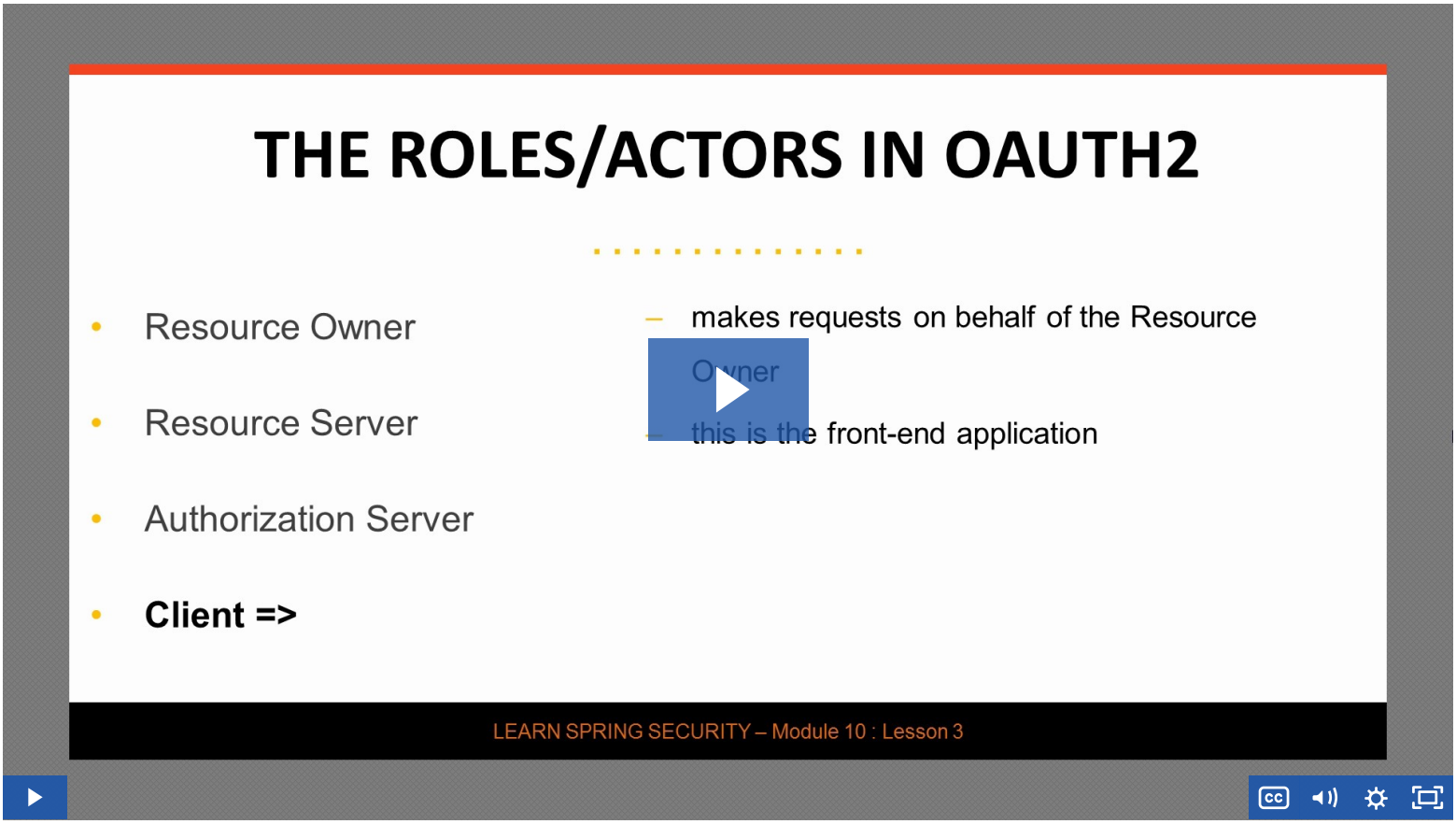
Next - Basic Authentication only covers - as the name suggests - authentication. Basically, you can't tell what permissions the user has because it has no concept of and no semantics for authorization.

Finally, there's no distinction being made between actual users and machines, so complex usecases are a no-go as well.

3. Resources

- Spring Security Basic Authentication

Lesson 3: How OAuth2 Works for REST - Part 1



1. Goal

The goal of this lesson is to introduce OAuth2 and give you a good understanding of the foundations.

2. Lesson Notes

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

2.1. What is OAuth?

Simply put, OAuth2 is a standard for authorization, more specifically, delegated authorization.

A high level goal of OAuth is allowing a Resource Owner to give access to a third party in a limited way, without giving away the master key (the password).

2.2. The Roles / Actors in OAuth

The Resource Owner (the user) is capable of granting access to a Resource.

The Resource Server (the API) is the host of the protected Resources.

The Authorization Server is capable of issuing Access Tokens to the Client.

The Client (the front end app) is capable of making requests on behalf of the Resource Owner.

2.3. Confidential and Public Clients

Confidential Clients are capable of maintaining the confidentiality of their credentials. For example - a server side client running on a secure server.

Public Clients on the other hand cannot guarantee they're going to be able to maintain the confidentiality of their credentials. For example - a native Android application, or an AngularJS client.

2.4. A Simple High Level Flow

- The Client asks for access from the Resource Owner - which grants it

- The Client talks to the Authorization Server

- The Auth Server gives the Client a key

- With the Key, the Client talks to the Resource Server and can access the Resource

2.5. Trust Zones

The Resource Server and the Authorization Server belong to a trust zone. The Authorization Server knows the what Resources to be protected are. And the Resource Server knows how to validate the tokens coming out of the Auth Server.

Then, obviously the Resource Owner and the Resource Server have a trust relationship with each other.

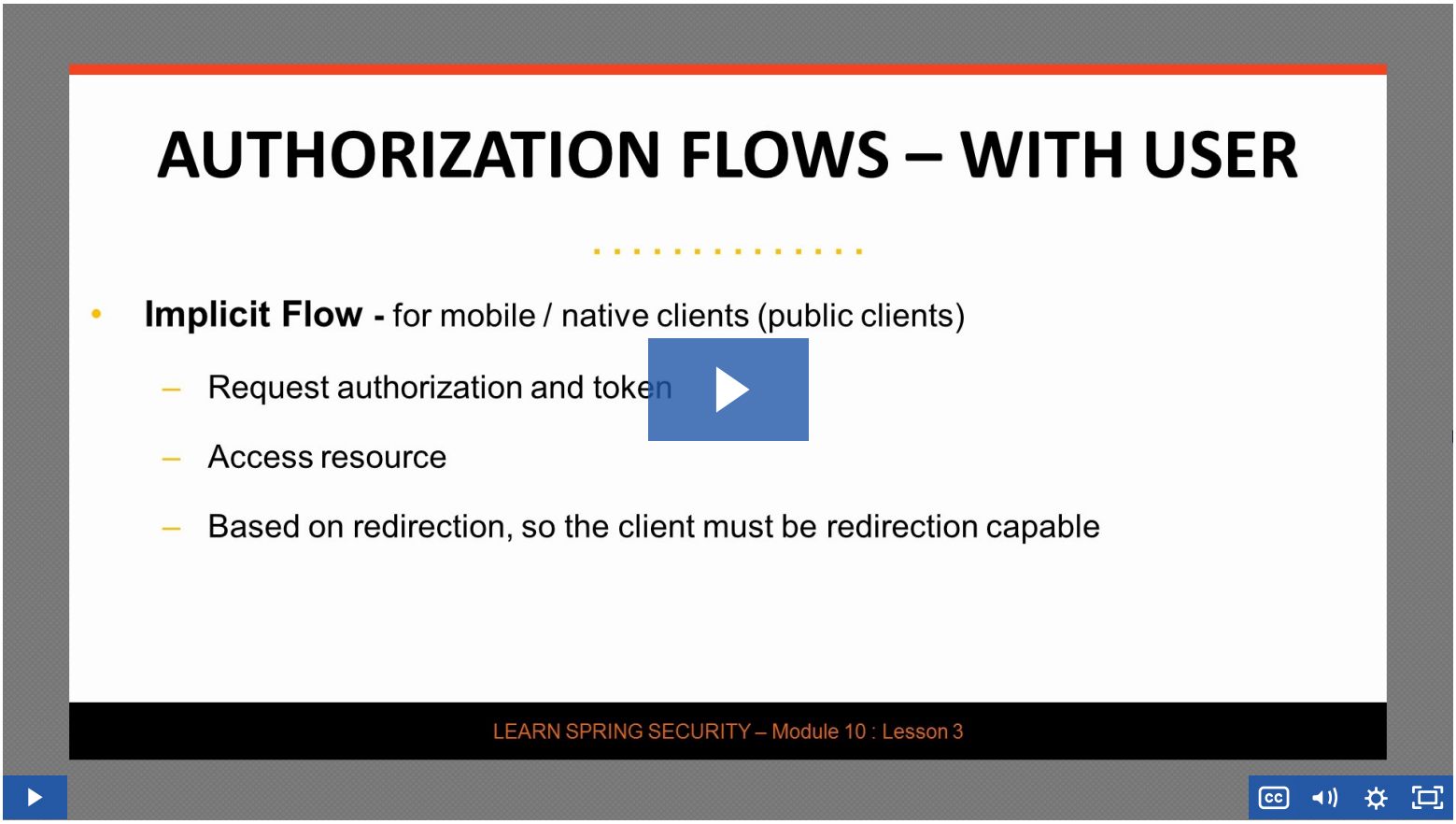
The Client doesn’t (necessarily) belong to any trust zone.

3. Resources

- About OAuth2

- An Introduction to OAuth 2

Lesson 3: How OAuth2 Works for REST - Part 2



1. Goals

The goal of this second part of the lesson is to explain and illustrate the standard OAuth2 flows.

2. Lesson Notes

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

We're looking at the 4 main OAuth2 flows here from the point of view of how they involve the user. The focus here is on separating the flows that involve user interaction from the ones that do not.

What's important to understand here is **what User interaction means** - not just using the credentials of the user, but actually involve a redirection and need the user to take action (review / accept permissions).

2.1. Authorization Flows - with User Interaction

The Authorization Code Flow - used for server rendered applications.

The Implicit Flow - used for Mobile / Native Apps (applications that run on the user's device). This is optimized for public clients and based on redirection (so the client must be capable of that).

A quick note is that the Implicit Flow has no refresh token - it gets the access token through the authorization request (not with a separate request as in the previous flow).

2.2. Authorization flows - with no User Interaction

The Client Credentials Flow - used for Server - Server communication, as there is no Resource Owner involved at all.

This flow is optimized for confidential clients and gets the access token using client credentials (client id and secret), not user credentials.

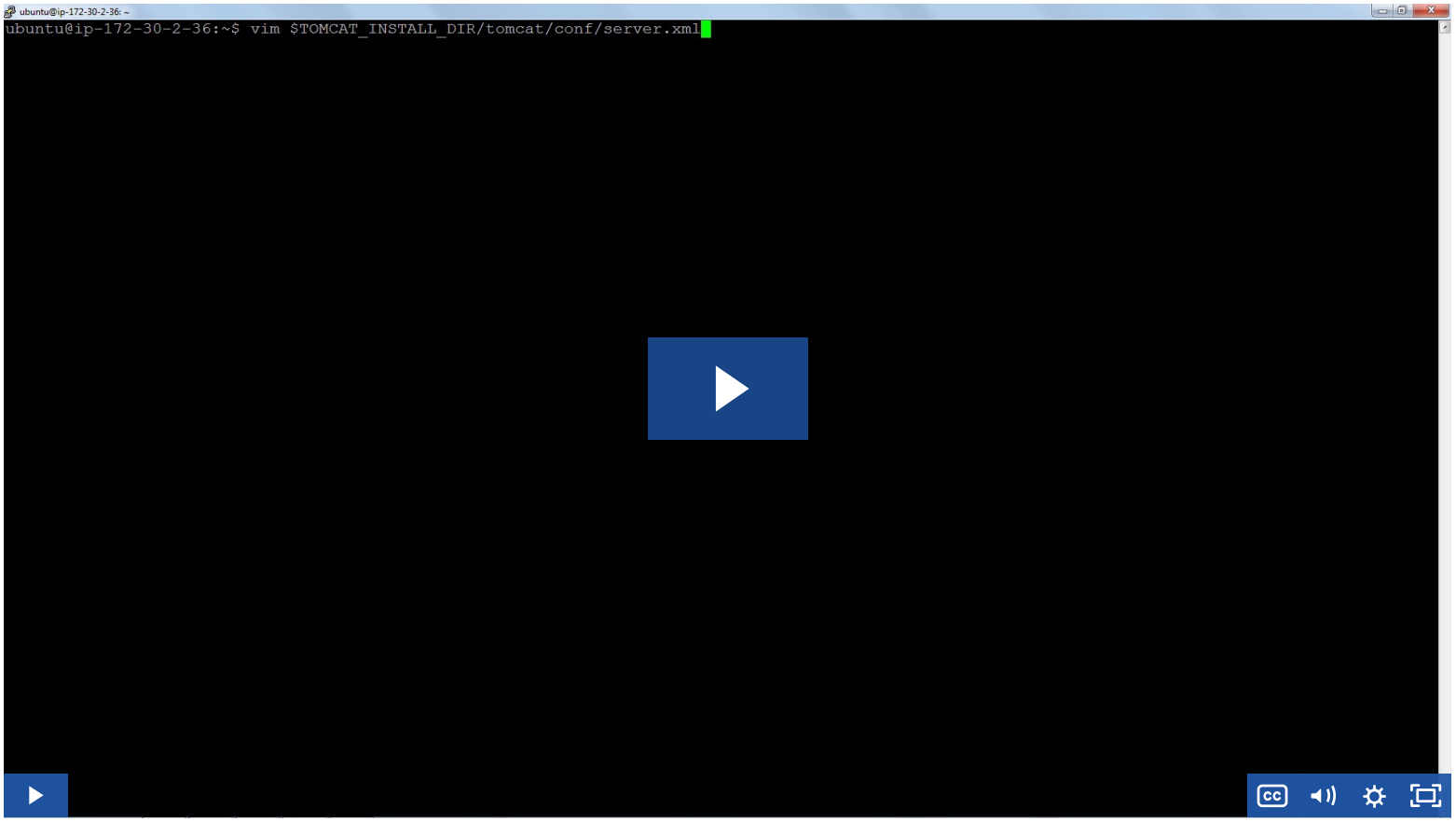
Resource Owner Password Credentials Flow - used for Trusted Applications (such as those owned by the service itself).

3. Resources

- About OAuth2

- An Introduction to OAuth 2

Lesson 4: Certificates and HTTPS for Tomcat



1. Goal

The goal of this lesson is to show you how to generate a self-signed certificate and use it to allow Tomcat to work over HTTPS.

2. Lesson Notes

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

Commands to generate the self-signed certificate:

```
/usr/lib/jvm/java-8-oracle/bin/keytool -genkey -alias tomcat -keyalg RSA
```

The result is the keystore file, under the home directory of the current user:

```
~/.keystore
```

We can also use *keytool* to verify the keystore that we just generated:

```
/usr/lib/jvm/java-8-oracle/bin/keytool  -list -keystore ~/.keystore
```

Finally, let's configure Tomcat with a new connector for HTTPS:

```
vim $TOMCAT_INSTALL_DIR/tomcat/conf/server.xml
```

Unable the HTTP connector and add these two lines to the HTTPS connector:

```
keystoreFile="${user.home}/.keystore"
keystorePass="changeit"
```

3. Resources

- SSL/TLS Configuration HOW-TOSSL/TLS Configuration HOW-TO