**Spring Security**

**Spring Security** is an out-of-the-box framework for Spring that ensures a secure Spring Boot Application.

Spring Security pre-configures easily customizable security functions to a Java application, and you can use it on any Java or Spring Boot application.

Essentially, Spring Security is a **library of security functions** you can add to your Spring, Spring MVC, Spring Boot, RESTful API, or single-page application on the Spring Framework. Also, integrating the Spring Security Framework is like importing a library of functions (and once again, you can add this library to any application on the Java platform).

**Once you enable it on Spring Boot**, it’s already **autoconfigured** to secure your application. However, based on the level of security you require, you can **customize what you need** to add or remove very easily.

You can use Spring Security for securing login in, ensuring that the right users have the appropriate level of access to your app, and so much more! 😉

your Spring application is made up of several **modules** that work independently of each other, and Spring Security is one of those. The **Spring modules**are like locked containers that allow other Spring modules to work together without getting in each other's way.

When you add **Spring Security,** its primary job is to safeguard the **HTTP requests** processed through the web application. You see, every time a user clicks a button or information passes from one part of the web application to another, it sends an HTTP request. These HTTP requests are what you need to secure.

These HTTP requests to your web application go through three levels of protection after adding Spring Security.

1. **HTTP firewall**
2. **Proxy design pattern**
3. **Filters**

Let's explore each of these more in-depth.

**Layer 1: HTTP Firewall**

The **HTTP firewall** is at the front line. A firewall is a wall that allows communication to go out but is very selective about what goes in.

It's like the bouncer at a nightclub, selecting who gets into the club based on specific criteria. In an application, a firewall blocks suspicious things that are coming in - usually, bad requests. It throws away any HTTP request that looks suspicious right off the bat.

**Layer 2: Proxy Design Pattern**

Second, comes the proxy design pattern. That's just a nerdy way of saying there's an outside delegating authority that will manage its access to a protected resource, which is your web application.

The Proxy checks and directs traffic

It essentially **classifies the HTTP traffic** and sends them to the appropriate servlet filters in the filter chain.

OK, awesome...but what on earth are servlet filters? Or a filter chain? 😦

Spring Security contains an **engine of servlet filters.** In Java, a **servlet** is a program that runs on your web application. The **filter**part is where Spring Security’s primary function lies.

Spring Security calls each little security configuration (or ruleset) a filter. So, *servlet* filters in Spring Security are just a bunch of small programs that perform a security function on your web application.

#### ****Layer 3: Filters****

**Filters** catch all of the **HTTP requests** that get into your web application and make sure they’re secure. Each filter provides a security configuration that you can add to your web application. A collection of these implemented servlet filters is called the **Spring Security filter chain**. 😄

So how does the proxy work with filters?

The proxy design pattern for Spring Security goes into a Java library of methods called the **DelegatingFilterProxy** class, where it invites Spring Security to your web application by including the filter chain in Spring Boot. The security filters are then added to create a **security engine** for the application.

What kinds of things can filters do?

Based on the complexity desired by the developer, it can provide **simple to advanced security configurations** against attacks. The Spring Framework also allows customizable servlet features required for the web application.

Even if you don't set up any filters, **your web application will still have a lot of great auto-configured Spring Security features just by adding it to your application!**

For example, say you have a filter that secures your login credentials. Once the HTTP call is made from the front-end and gets through the first firewall, **DelegatingFilterProxy** will classify and send the HTTP traffic through the filters that deal with your login credentials in the Spring Security filter chain, so the proper controls are in place.

So to sum up, Spring Security secures your web application’s HTTP requests by passing them through three layers:

* First, the **HTTP firewall** blocks suspicious requests.
* Second, the **proxy design pattern** (DelegatingFilterProxy) will take the rest of the HTTP requests and send them to the **Spring Security filter chain**.
* Finally, the**filters** in the chain will **take those HTTP requests and make sure that they follow the security rules in the filter.**

So that's all excellent theory, but there's no better practice than setting it up yourself. Let's get started by making a new Spring Boot app and installing Spring Security.

### Create a Spring Boot App With Spring Security

We’re going to create a new **standalone Spring Boot application.** This will gives you the baseline app you need to work with Spring Security throughout the rest of this course.

If you are securing a web application that has been created by Spring Boot, it is easier to integrate Spring Security than with non-Spring Java applications. You can also integrate Spring into **Maven projects or any Java EE project** by creating an XML configuration file.

Integrating the Spring Security framework into a Spring Boot app is similar to importing a library of functions.

#### Use Spring Initializr to Create a Spring Boot App With Spring Security

First, set up a standalone **Spring Boot** web application. The **Spring Initializr** (yes, I spelled that right) is a**GUI** interface you can use to configure your **Spring Boot** application. The great news is that **Eclipse** is already equipped with **Spring Initializr** to make creating your Spring Boot app easier.

If you don't have Eclipse, check out my [**Set up your Java Environment**](https://openclassrooms.com/en/courses/5684376-set-up-your-java-development-environment) course. If you are not using Eclipse, you can use the Spring Initializr from this [**website**](https://start.spring.io/)  to **create a new project.**

### Let's Recap What We Did:

* Click on File -> New -> Project.
* Look for the **Spring Boot wizard.**  You can scroll or type it in the search box.
* If you don't find it, go to Help -> Eclipse Marketplace.and search for the latest **Spring Tools release.**  This includes **Spring Boot** support. Click ***Install,*** and restart Eclipse.

Now, on to the next phase:

[[Graphical user interface, text, application, email

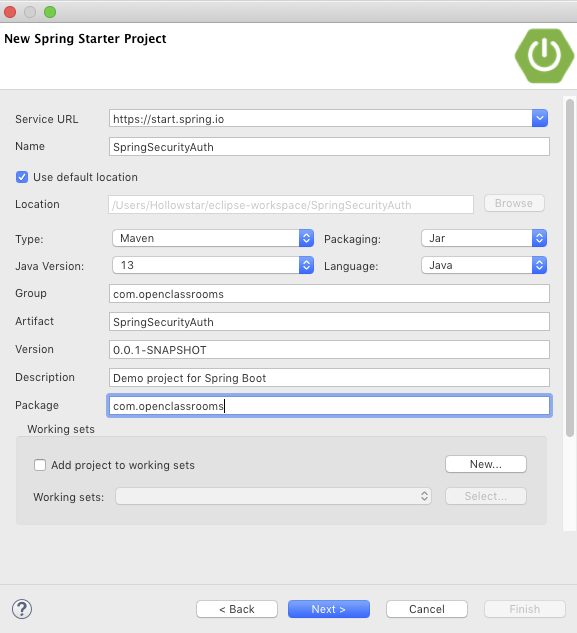
Description automatically generated](https://openclassrooms.com/en/login?context=courseAccess)](https://openclassrooms.com/en/login?context=courseAccess)

* Choose the **Spring Starter Project** to start the **Spring Initializr,** and click Next.

New Project wizard

 You will see that the wizard connects directly to the **Spring Initializr** URL. Now fill out the following:

* **Name:** Choose a name.  I called it SpringSecurityAuth.
* **Java Version:** Choose the latest according to the JDK version you have installed on your computer.  It defaulted at 8, and I chose 13.
* **Group:** I will call it com.openclassrooms.
* **Package:** To match my group name, I called it com.openclassrooms.

Spring Initializr in Eclipse

After you click Next, you will add your **dependencies,**whichis an essential part of the project. Add the following dependencies by typing them into the search bar and checking the box next to the name. Add these three dependencies:

* Spring Web
* Spring Security
* OAuth 2 Client

Graphical user interface, application

Description automatically generatedAdd dependencies

Click Finish when you complete it. Now you can grab a cup of Java as you wait for Eclipse to download your resources!

Download progress: Bottom Right on EclipseDownload progress: bottom-right on Eclipse

Click on Window -> Show View -> Project Explorer to see the Spring application project. If you click on the down arrow, you can see the file structure for your Spring app.

It's good practice to ensure that you have all the proper updates on Eclipse.  You can check by opening Eclipse and click Help -> Check for updates.

#### Manage Your Dependency Hierarchy in Spring

Thanks to **Spring Boot,** the**Spring Initializr** made a nifty little app, including the necessary files, so all you have to do is add your files, and you'll be on your way.

The Spring Framework runs on several different **modules,**whichare different types of functionality that help Java developers pick and choose what they require for the Spring app. It also makes it easier to integrate other frameworks. For example, Spring Security is a module and includes Java classes used to implement the security functionality you will be using. The dependencies are the Java libraries that your Spring Boot app needs to run its functionality.

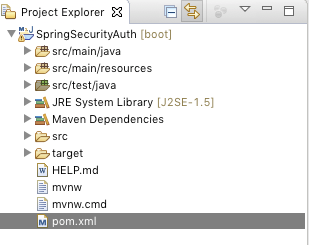
So, now you have created a Spring app using **Spring Boot** and added three dependencies to it on Spring Initializr: Spring Security, Spring Web, and OAuth 2 Client.

Wait, so why did we add those dependencies?

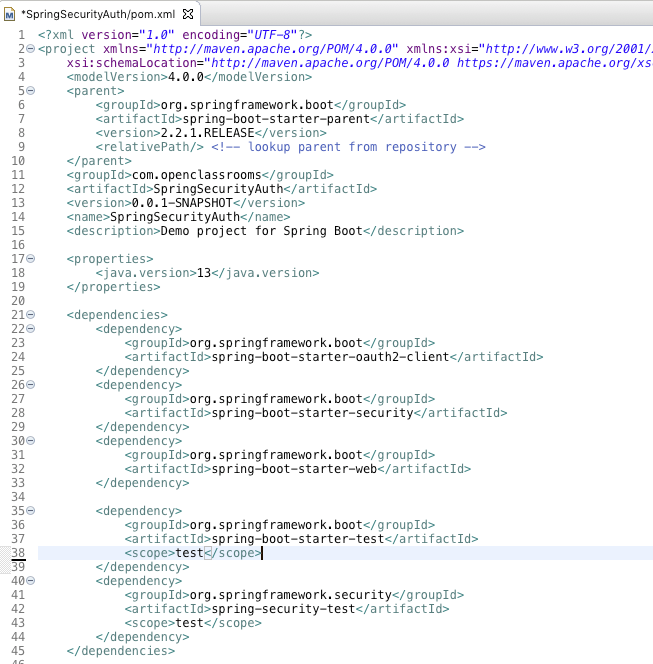
We added:

* **Spring Web** to give you web layer functionality, hence making it a web application.
* **Spring Security** to give you the security layer that begins doing its job before you even initialize the application.
* **OAuth 2.0 Client** because you will use its special security protocol on your login form. We haven't talked about this, yet, but you'll use it in Part 2 of this course.

Let’s take a look at the dependencies.  If you look in your **Project Explorer** on the left pane in **Eclipse,** you will find the **pom.xml** file in the target folder on the bottom here:

Project Explorer view

When you open up the **pom.xml** file, you will find your dependencies. A few more came implicitly with Spring Initializr. If you look carefully, they are dependencies used for testing.

Dependencies in pom.xml

You may notice something interesting.  **OAuth 2.0 Client** is the first dependency listed, followed by Spring Security and Spring Web.

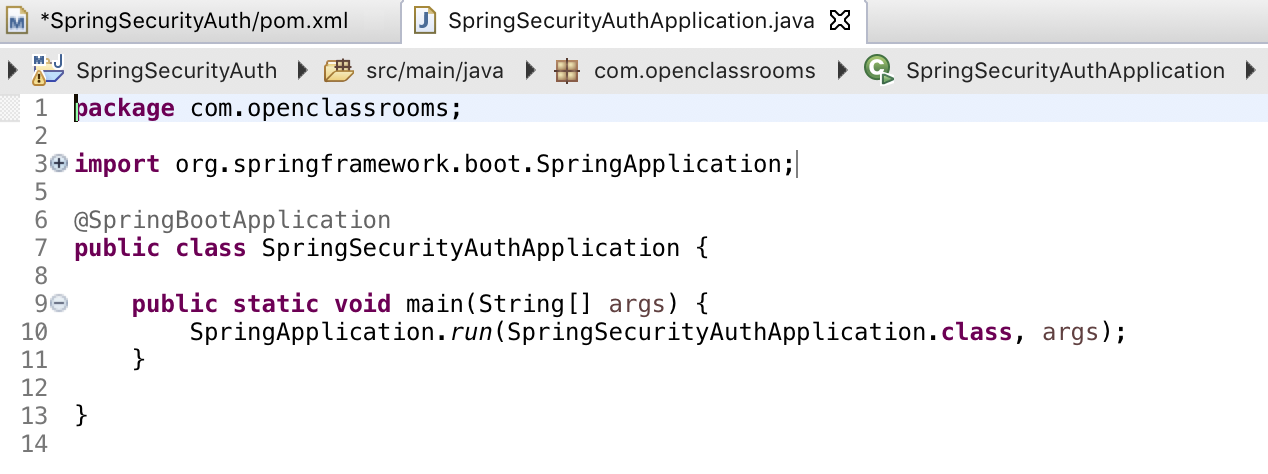
Why is that?

Well, this is called the **dependency hierarchy.** The Spring Boot application requires that certain dependencies need to be injected and initialized before other ones.  In this case, OAuth 2.0 is a foundation for security rules that will be added automatically, followed by the in package rules of Spring Security.

If these dependencies are initialized out of order, they will not work properly.  If you add the dependencies in the **pom.xml** manually, ensure that they are placed in the proper order from top to bottom.  The same is true of your filters in Spring Security.

First, you want to open up your main class in src/main/java. It should be under your package name.  Mine is called  com.openclassrooms.

If you called your application **SpringSecurityAuth,** the main class will be called **SpringSecurityAuthApplication.** When you open it up, it will look like this:

Your Spring Boot main class file SpringSecurityAuthApplication.java

This will be your **main  method** where your Spring Boot application opens up and starts.

### Let’s Recap!

In this chapter, you learned the ins and outs of the Spring Security Framework.  Here are some of its main features:

* The **Spring architecture** works with Spring Security to add three layers of security to your web application:
  + **HTTP firewall-** catches the suspicious traffic coming to your web application.
  + **DelegatingFilterProxy-** routes the rest of the HTTP traffic to its designated security filter.
  + **Security filter chain-** the main security configuration that houses the security rules for the web application.
* **Spring Initializr** is a tool used to autoconfigure a Spring Boot web application.
* **Dependencies** are the main code libraries that you can import into your web application.
* We added **Spring Web, Spring Security,** and**OAuth 2.0** dependencies.

To use Spring Security, you need something to secure. Since, at one point or another, as a Java developer, you’ll have to obtain a ***login form*** and secure it, we’ll practice working with that in the next chapter.

When you go to your bank’s website and log in, you assume that you are the only one with access to your account information, right? You wouldn’t want someone else to be able to log in as you or find a way to use your account.

When you log into the bank, the web application **authenticates** that you are you. 😄 When you look at a page with your account - for example, your bank statements - the application is **authorizing** you to look at a particular page. These two features - **authentication and authorization** - are at the heart of **Spring Security.** If you can keep this process secure, you’ve won half the battle.

In security, there is a term called **access control,** and it essentially controls **access**toa web application that requires a login mechanism.  **Access control** comes in two steps. Can you guess what those are?

**Authentication and authorization?**

Yes! We’ll be using Spring Security to implement access control into your Spring Boot app, but first, let’s dive into the details of authentication and authorization.

### **Define Authentication**

When you go to an office building for a large corporation, you typically see guards checking Ids or a machine that scans your badge. This is a **physical access control** feature that ensures that you can prove whom you are before going inside.

In web applications, how do you think you authenticate to the site and prove that you are who you are?

**By logging in with your password?**

Right! Typically in web applications, you authenticate by logging in with a username and password. A username identifies who you are. The password is something only you know. This is called **single-factor authentication**because you only have to know one way (the password) to prove who you are.

Sometimes you provide additional elements, like biometrics, a physical token, or a badge. This is called **multi-factor authentication** because you have to (1) know your password and (2) have something else to prove you are whom you say you are.

**Once you are logged in, the web application already authenticated you because you have proven that you are you. Here comes the next step: authorization**.  😁

#### Choose From Multiple Types of Authentication

Let’s talk about different kinds of authentication. You learned about **single-factor authentication** where a user logs in with a password. You also learned about **multi-factor** authentication, where a user can use multiple items to prove identity. There are also different types of ways your web browser handles your authentication:

##### **Session-Based Authentication**

In **session-based authentication,** users first log in using their credentials. When they are successful, they have authenticated themselves and have started a **session.** **A session is the time between when the user logs in and logs out.** The server saves the user’s session information and sends a copy of it in a small file (a cookie),also saved in the user’s browser. Typically, this information may include the user’s credentials, how long the session lasts, and a **session Id** number. Every time the user sends a request through the web application, the server checks with the cookie to ensure the session credentials match those on the server and are still valid.

Because there's a copy of the user's credentials on the server and the cookie, **session-based authentication** is called **stateful.**

##### **Token-Based Authentication**

 In **token-based authentication,** the user authenticates to the server, and the server sends the session information in a small file (a token) to save only on the user’s browser or local computer. A token is similar to a cookie in function, but there are some significant differences. **With session-based authentication with a cookie**, the user’s credentials are saved on the server’s database as well as the user’s browser. In token-based authentication, a **JSON Web Token (JWT)** has the information needed to validate the user, so the session information does not need to be saved on the server, making it **stateless.**

So what exactly is this **JSON Web Token?**

**JSON** stands for **JavaScript Object Notation.** The JSON Web Token is a little **JavaScript object** that encrypts and transmits your authentication information. It validates that a user is authenticated because it holds **encrypted information only the authorization server can understand**. It makes things more efficient and secure than using a **cookie**, so most Java web applications are based on **JWT.**

### Understand Authorization

To understand the **difference**between authentication and authorization, let’s go back to the office building analogy. Say you’ve already made it inside after **authentication.** You are now in the building and would like to go to your department. Your office is on the fifth floor with all the other developers. You go to the elevator and hit the button for the fifth floor, but it doesn’t light up. 😕

You then remember you have to scan in your badge before you can select your floor. You see, are **authenticated,** but now there are checks inside the building to see if you are **authorized** to access other parts of the building.

**Authorization** happens after **authentication.**  The web application takes **another** measure to see if you are allowed access to certain parts of the web application.

Applied to web applications, we typically have **user** and **administrator** scenario. When an administrator is authenticated into a site, the web application allows that person a different set of rights than a regular user. This is where **authorization** comes in.

It’s not just people that need authentication and authorization. Sometimes you are protecting a **network, application, networked device, computer,**or **server.** In these cases, you see **processes** (computer updates run in the background with administrator credentials), **resources** (your computer wants to access a file in a server or get on a network), and **systems** (servers, computers, network devices, etc.) that need to authenticate and be authorized.

What would this look like in a real situation?

Imagine that the sales department in a company has a computer that’s not locked down. They handle transactions with outside entities, so their computers and network are not as secure as the rest of the departments in the company. These computers are only allowed in the sales department section of the company’s network. So, the network is set up so that when the computer authenticates into the network, it is only authorized to be used in the sales department network. The computer cannot be used in the more secure parts of the company’s network.

Let’s go over what you’ve learned about different ways people get authenticated and authorized on web app login forms.

### Let’s Recap!

* **Authentication** checks to make sure that the user knows the right information to prove they are who they say they are.
* **Authorization** makes sure that the authenticated users only go to the pages they are allowed to see.  Two types of authentication are:
  + **Session-based authentication**uses cookies to store information about the user’s session. The cookies are stored in the user’s browser and on the authorization server’s database.
  + **Token-based authentication** uses a JWT sent from the authorization server. The JWT is used to validate that the user is logged in and can be stored in the browser, but is best stored in a secured token on the browser.

Let’s apply what you’ve learned in this chapter and create a default Spring Security login page in the next chapter. We’ll also configure a simple Spring Security filter chain to define its security configuration. Let’s get to coding!

**Authorization** is a safeguard to ensure you’re accessing the pages you are supposed to. If you’re logging in as a regular **user,** you shouldn’t access an **admin** page. Well, the way to set up who can access what page is by creating **roles.** You would create a user role to access user pages and the admin role to access admin pages. This is called **role-based access control** because you’re controlling access to different parts of a web app based on the roles of the people using it!

**Access control** is a crucial subject area and is used for both physical and software security. Access control is essentially setting up at least one safeguard to prevent access to the wrong people. Like a security guard checking badges, you use it when you log into a website to access your account. Whether you need a badge or a password to get in, it all falls under access control.

 Guess where you manage the roles in your web application?

The Spring Security filter chain.Let's make it happen!

### Set Up Your Spring Security Filter Chain

[[Graphical user interface, text, application, email

Description automatically generated](https://openclassrooms.com/en/login?context=courseAccess)](https://openclassrooms.com/en/login?context=courseAccess)

Now that you understand authentication and authorization let’s use it on your **Spring Boot web app with Spring Security.**  Right now, you are going to use it to create a login form.

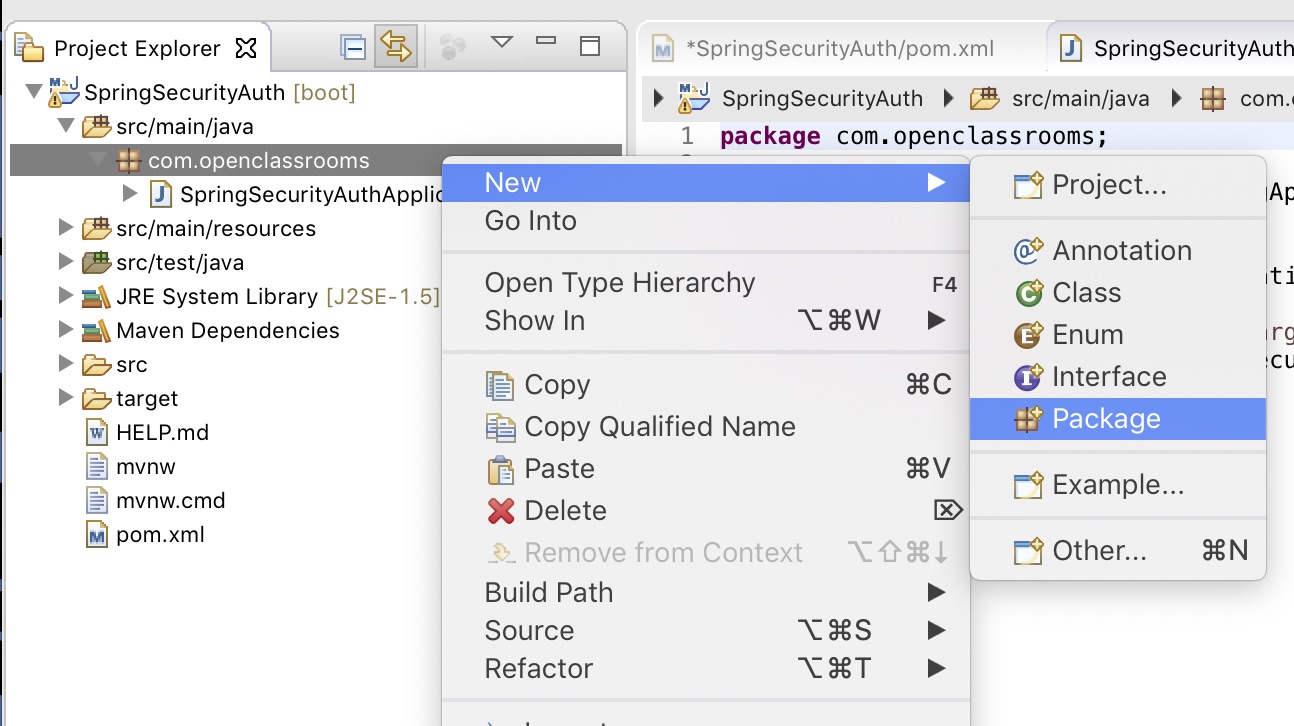
Spring Security offers a method called  loginForm()  to create a default login page, so you don’t have to create anything. When setting up the custom filter chain, there is a bit of manual coding, but not much, and it’s easy to add and remove different methods.

First, you need to create a file for your new super impressive custom-built Spring Security configuration. So far, you have your main class, **SpringSecurityAuthApplication.java,** and you should have that in your **src/main/java** folder.

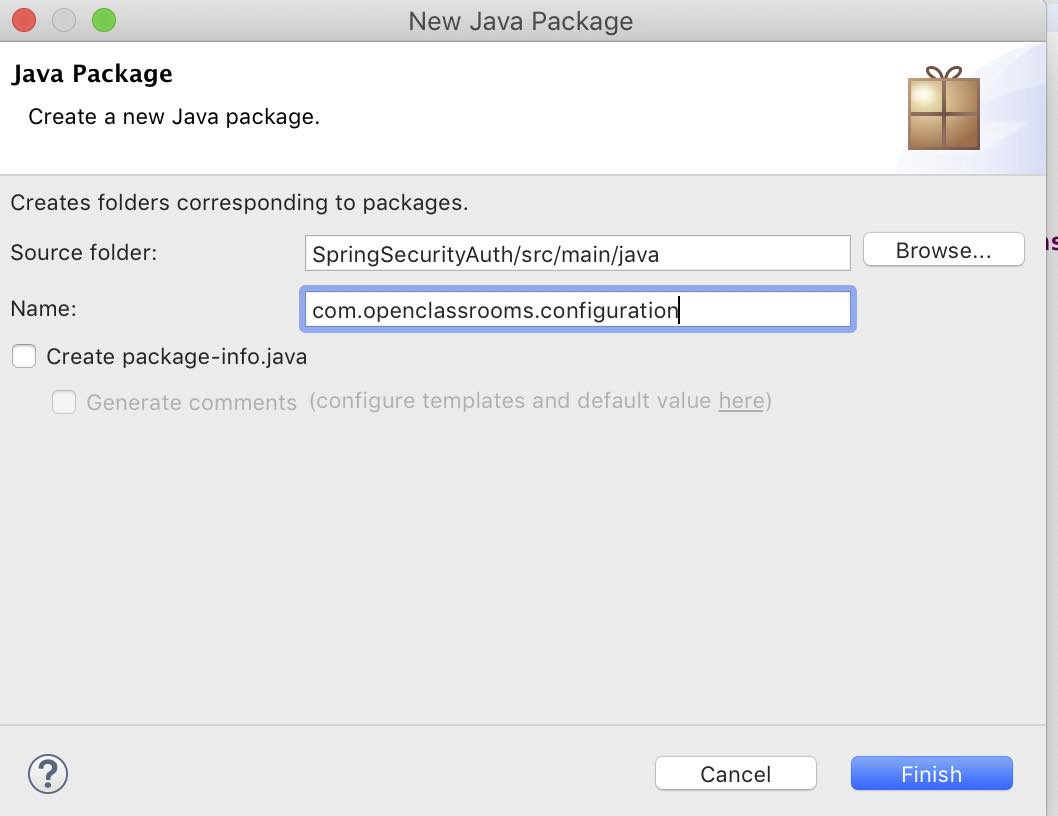
Let’s create a new class for your **Spring Security configuration** file in the same folder that holds the **SpringSecurityAuthApplication.java** file.

When you build a web app in **Eclipse,** it follows a **hierarchy.** The best way to make sure that your class with the main method runs **first,** create child packages under your main package. So if your main package is  com.openclassrooms, your other classes go in packages built on that.

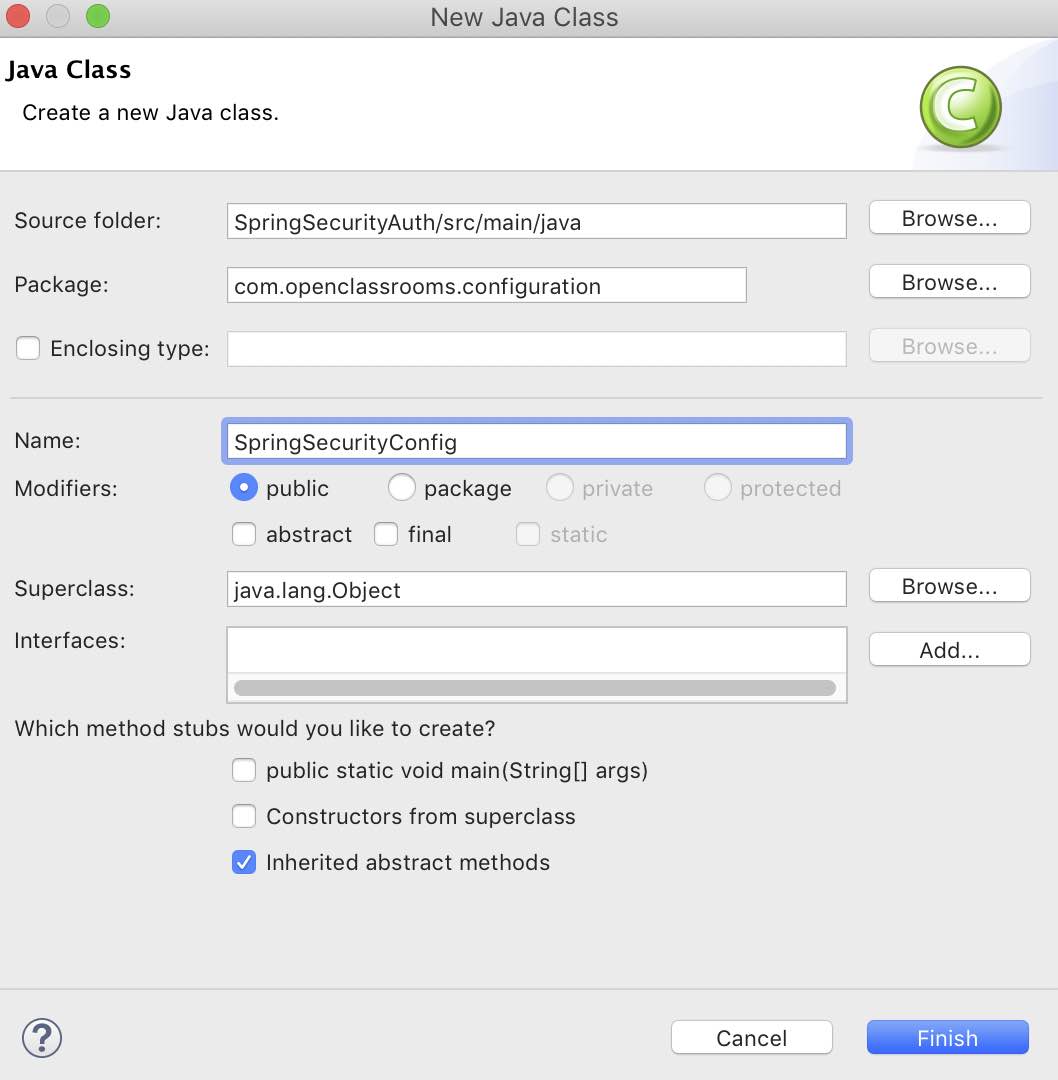
Right-click on the the  com.openclassrooms  package for your java files and create a new package.

Create a new package

Add the configuration extension on the package name so it will now be called  com.openclassrooms.configuration.

Create a package called  com.openclassrooms.configuration

Create a new class in that package called**SpringSecurityConfig.**Right-click on the  com.openclassrooms.configuration  package -> New -> Class:

Create SpringSecurityConfig class

Let’s be creative and call the new file SpringSecurityConfig. 😎 The only thing you need to add is the class name, so put in SpringSecurityConfig  for the Name. You can leave the defaults as they are for a new public class, and click Finish.

Now you should have a new class called **SpringSecurityConfig** to start your custom web configurations for a secure web application. Let’s begin with the fun stuff!

### Build the Security Filter Chain

[[Graphical user interface, text, application, Word

Description automatically generated](https://openclassrooms.com/en/login?context=courseAccess)](https://openclassrooms.com/en/login?context=courseAccess)

Now that you are good to go with the new class, let’s use Spring Security’s powerhouse class to initiate your very own custom security filter chain. Remember what **security filter chains** are? They are the file in which you add and remove security filters based on what you need for your web application.

Make sure that you have your filters in the right order because some may need to be used before others to work properly.

Follow these steps to build your security filter chain:

* Add the  **Configuration** and **EnableWebSecurity** classes to designate the **SpringSecurityConfig class** as the security filter chain configuration class for the Spring Boot app and to create the security filter chain, respectively.
* Add the  configure()  method with **AuthenticationManagerBuilder** class to handle the authentication ruleset.
* Add the  configure()  method to the **HTTPSecurity** class to push all HTTP requests through the security filter chain and configure the default login page with the  formLogin()  method.

Now let’s take a look at each of these steps in detail.

##### Add the Configuration Class

Now that you know what this class does for your Spring app, let me show you how to implement it in your **SpringSecurityConfig.java** file. You don’t want your compiler to scream at you, so import the required library for this class to work:

import org.springframework.context.annotation.Configuration;

Then add **@Configuration** to your web applic ation above your class declaration.

##### Add WebSecurityConfigurerAdapter

Now extend your SpringSecurityConfig class with the WebSecurityConfigurerAdapter. This designates the SpringSecurityConfig class as a Spring Security configuration. To use WebSecurityConfigurerAdapter, add another import:

**import** org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

Extend WebSecurityConfigurerAdapter to your SpringSecurityConfig class like this:

@Configuration

public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

}

##### Add EnableWebSecurity

Now add EnableWebSecurity as your final step. This will ensure that the Spring web application knows to import your custom Spring Security configuration. So guess what you need to do to add it? An import for the EnableWebSecurity libraries:

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

Next, add  **@EnableWebSecurity**  above your class declaration.

@Configuration

@EnableWebSecurity

public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

}

Now use Spring Security’s configure()  method to hold your actual security filter chain.  It's the meat of it, so pay attention!

##### Add HTTPSecurity

To add some functionality, create a login page with buttons. You can do that in the Spring Security configuration file you are making with **HTTPSecurity.** This class is invoked to run the security filter chain on HTTP requests. By default, it runs security on all of the HTTP requests unless specified in this security filter chain.

This is where you build your security filter chain. So first, guess what you add? The library!

**import** org.springframework.security.config.annotation.web.builders.HttpSecurity;

Take a look at this code here. You can start with  configure()  method for all HTTP requests that are the inputs:

@Override

public void configure(HttpSecurity http) throws Exception {

}

So here’s an extra special security bonus for you: if you add  @Override  before the  configure() method declaration, you can override the defaults that are normally configured into the app with Spring Security.

Why would you want to do that? That sounds like double the work.

Well, this makes it more difficult for a hacker to **fingerprint** your Spring Security app! The less a hacker knows about your app, the better. It looks something like this:

package com.openclassrooms.configuration

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

@Configuration

@EnableWebSecurity

public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

}

}

##### Default Security Headers for Spring Security

When you add the Spring Security dependency, it comes with a default configuration that makes it pretty secure out of the box.  Let’s take a look at what it has:

Cache-Control: no-cache, no-store, max-age=0, must-revalidate

Pragma: no-cache

Expires: 0

X-Content-Type-Options: nosniff

Strict-Transport-Security: max-age=31536000 ; includeSubDomains

X-Frame-Options: DENY

X-XSS-Protection: 1; mode=block

Alright, so what does all this mean?  Let's walk through it:

* **Cache-Control -** if you’ve authenticated onto a site, your browser may save a cached copy, potentially allowing a malicious user to view the cached page even after you’ve logged out. Cache-control prevents the storage of a cached copy of that page.
* **Content-Type** **-** the  nosniff  flag means that a malicious user cannot guess the content type of a request or execute a random **XSS (Cross-site scripting)** attack. Content-type is the type of file and can be a PDF, mp3, Docx, etc.
* **HTTP Strict Transport Security (HSTS)** **-** means that anyone who types the URL in the box is automatically taken to the HTTPS version of the website. Using HTTPS throughout your site for encrypted querying is recommended practice.
* **X-Frame-Options** **-** Many sites have iframes, which allow a hacker to hijack your site by adding a URL to a malicious host server, download files onto your browser, and access your computer through exploit kits. Typically, a victim of **clickjacking** accidentally clicks on the bad URL and gets pwned! The  DENY  flag prevents this sort of clickjacking attack
* **X-XSS-Protection -**  mode=block  blocks malicious JavaScript from executing on your site and therefore prevents suspected XSS attacks.

These defaults are pretty significant. However, you may need to customize or remove them altogether. You can change the default security headers in your  configure()  method if you use  @Override.

With that said, let’s build a Spring Security login form using these out-of-the-box Spring Security features described above. You’ll learn how to customize some of the features later in the course.

Now, let’s look at how you can invoke the default login form with Spring Security.

### **Configure a Custom Login Page With Spring Security**

Now let’s configure your Spring Security filter chain for authentication and authorization. You can add a few filters to this function block to make this page do something useful. Start with a default Spring Security form using the  formLogin()  method.

@Override

public void configure(HttpSecurity http) throws Exception {

http

.formLogin();

}

##### Add a User and Admin Role to Your Filter Chain

Spring Security provides **filters** that can be used to authenticate different types of **roles.** Give yourself two roles in your Spring Boot app:  **user** and **administrator.** Once they authenticate, Spring Security plays a crucial role in handling authorization for controlling access to data based on roles.

First, add the  authorizeRequests()  method to set up the roles after it.  Then add the  antMatchers()  method to provide a mapping for the  USER  and the  ADMIN  roles. This ensures the **authorization** portion of the landing page for your users. The  ADMIN  role gets a homepage specific for admin, which is  /admin, but also has access to  /users, and the  USER  role gets the  /users  homepage and should not have access to the  /admin  homepage. Add  anyRequest().authenticated()  so the form below will be used for authentication.

Do this by adding this code snippet to your  configure()  method that handles the HTTP requests.

@Override

public void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers("/admin").hasRole("ADMIN")

.antMatchers("/user").hasRole("USER")

.anyRequest().authenticated()

.and()

.formLogin();

}

OK, so now you have the security filter chain for your HTTP requests taken care of with HTTPSecurity. Let’s create a security filter chain for the authentication portion of your masterpiece. You do this with AuthenticationManagerBuilder.

##### **Add AuthenticationManagerBuilder**

Let’s try out a simple login page to test out role-based access using Spring Security. Use the **AuthenticationManagerBuilder** to create a user and an admin, and assign them to roles. This filter not only creates encrypted credentials, but it also sets them up with roles.

Next, you see  auth.inMemoryAuthentication. This means that created credentials are stored in memory, rather than a token or database like JDBC.

If you want authentication with a connection to a database, you can set up JDBC based authentication the same way. Instead of using  auth.inMemoryAuthentication, use auth.jdbcAuthentication() and add a database configuration. Here’s some great info on how to [**set it up**](https://www.baeldung.com/spring-security-jdbc-authentication).

Here is the library for **AuthenticationManagerBuilder.**

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.inMemoryAuthentication()

.withUser("springuser").password(passwordEncoder().encode("spring123")).roles("USER")

.and()

.withUser("springadmin").password(passwordEncoder().encode("admin123")).roles("ADMIN", "USER");

}

I set up a user and admin you can log in with. These are the credentials:

* **USER role**: user: ***springuser***password: ***spring123***
* **ADMIN role**: user: ***springadmin***password: ***admin123***

I have also added encryption for the passwords.  That means that even though I have hardcoded in some easy passwords, the  encode()  method will encrypt them.

As the last method in the **SpringSecurityConfig** class, add a **bean** with the  passwordEncoder() method. The cool thing about the **PasswordEncoder** class is that you can choose the type of hash algorithm you want to encrypt your password. For this example, I chose  **BCrypt**because it is one of the best-rated encryption algorithms out there for password encryption. You should use it as well.

If you'd like to check out other hash algorithms you can use with this class, check out the **[PasswordEncoder class page](https://docs.spring.io/spring-security/site/docs/4.2.12.RELEASE/apidocs/org/springframework/security/crypto/password/PasswordEncoder.html" \t "_blank).**

To add this functionality, add three libraries to your imports:

import org.springframework.context.annotation.Bean;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

Add the following code in your **SpringSecurityConfig** class:

@Bean

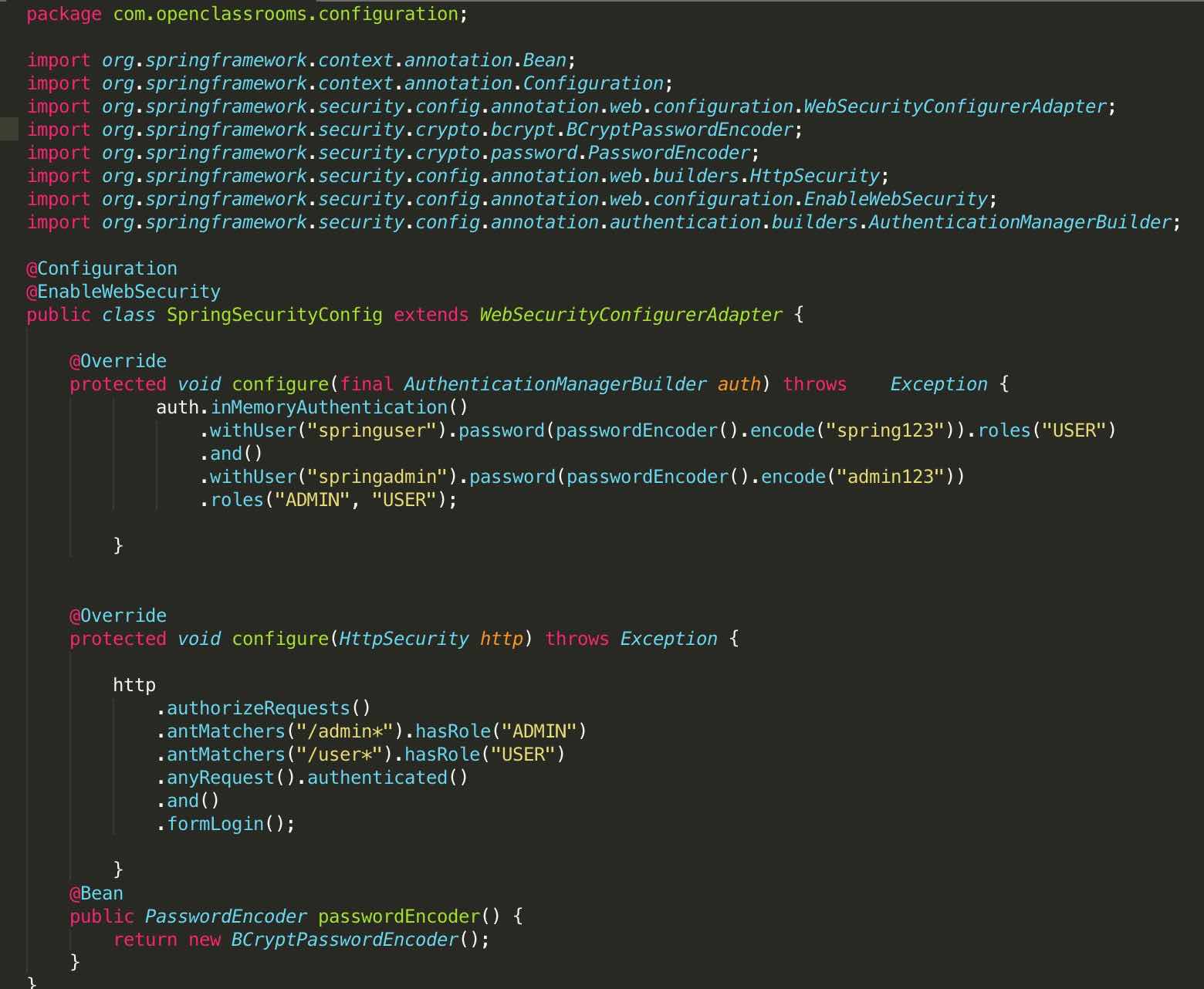
public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

##### Review Your Security Filter Chain

Let’s go over the **SpringSecurityconfig.java** file you completed. It should look a little like this if you followed along - pay attention to the formatting: 🙂

SpringSecurityConfig

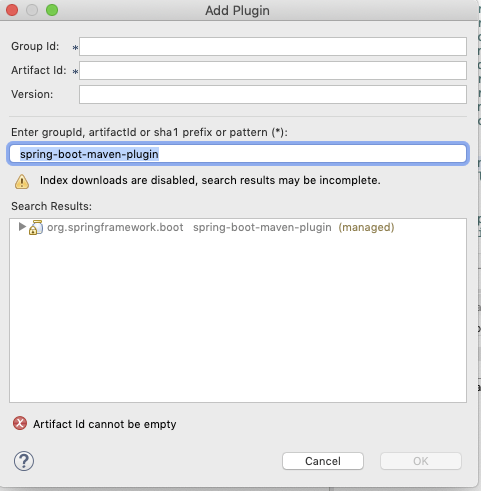
### Set Up Your Controller

[[Graphical user interface, text, application, email

Description automatically generated](https://openclassrooms.com/en/login?context=courseAccess)](https://openclassrooms.com/en/login?context=courseAccess)

Next, you're going to set up a **REST controller** to create your homepages for you. To make sure your **REST controller** will work, ensure you have updated **Maven** and have the **Maven plugin** as your dependency. Eclipse makes it easy to do just that:

* Go find that **pom.xml** file that holds your dependencies.
* Right-click on pom.xml -> Maven -> Update project.
* You also need to add the Maven plugin for additional web support.
* Right-click on pom.xml-> Maven -> Add Plugin.
* Type in  spring-boot-maven-plugin.

Add Maven plugin

Once you add the plugin, it should show in your **pom.xml** file. It should update the libraries you need to make your **REST controller**work. It is also vital to create a new package for your controller. It ensures that your web application knows where to find it.

* Right-click on your package ***com.openclassrooms***-> New -> Package.
* For the name of the package, add the word controller to your main package name.  Mine is called**com.openclassrooms.controller.**
* Now, right-click on the package you just made **c*om.openclassrooms.controller -***> New -> Class.
* Name your controller class **LoginController.java.**
* Add the  @RestController  annotation for library support of your REST Controller.

Then create a different method for each role using the  @RolesAllowed  class and  @RequestMapping  for the URL mapping to that role:

package com.test.controller;

import javax.annotation.security.RolesAllowed;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class LoginController

{

@RolesAllowed("USER")

@RequestMapping("/\*")

public String getUser()

{

return "Welcome User";

}

@RolesAllowed({"USER","ADMIN"})

@RequestMapping("/admin")

public String getAdmin()

{

return "Welcome Admin";

}

}

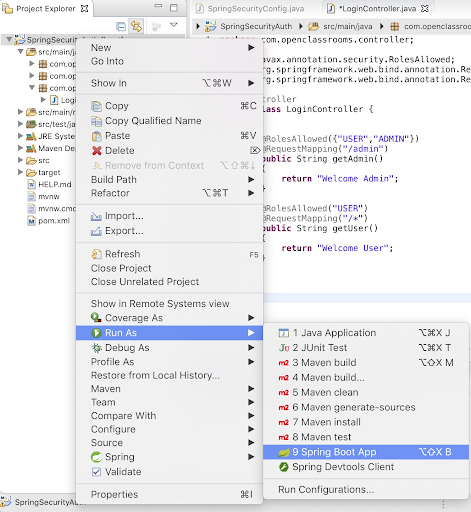
This **REST controller** is pretty awesome. Instead of having to create a whole new homepage for the admin and the user, at the  /admin  and  /user  URL, the controller dynamically creates the pages for you. Easy peasy!

So now that you’ve set up all this stuff, let’s look at your handiwork on the browser.

##### Set Up Your Web Server

When you run your standalone Spring app, you can pull up your login page on your browser using **Apache Tomcat** as your web server. There's no configuration needed; this happens automatically when you compile and run your code.

* **Step 1:** On Eclipse, go to the folder in your Project Explorer that shows you the name of your app. It will say [boot] after it in brackets. Click Run As -> Spring Boot App.

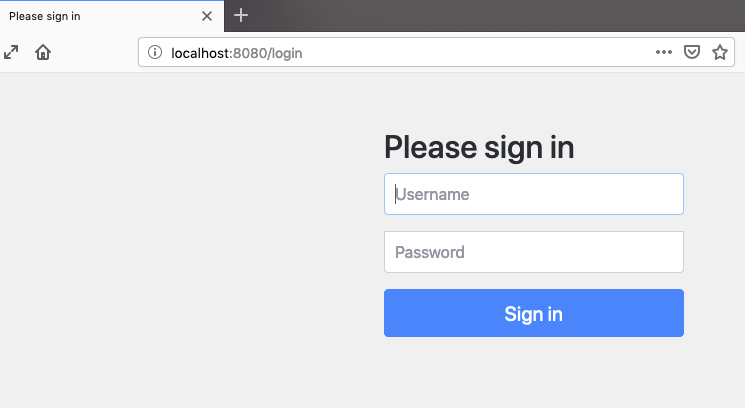
Run SpringSecurityAuth app

You can see your code compile in the console, and once it is complete, you can open up your web application in the browser.

* **Step 2:** Open up your favorite browser and type in this URL:  [http://localhost:8080](http://localhost:8080/).

If you want to change the port you are using to something else, like 8090, you can put this line, in your**application.properties** file under **src/main/resources.**

Now log in as the user to see what happens:

Default Spring Security login form

Alright, it works!  Here are some additional things to try out to make sure you set it up correctly:

* What happens when you put in the wrong password?
* What happens if you try to go into the /admin page when you are logged in as a user?

If you run into any issues, check all of your code and dependencies to ensure you have everything set up correctly.  Eclipse is pretty good at finding errors so you can work through the issues as they come.

Be sure to close the connection to your  localhost:8080  when you are done; otherwise, the next time you compile and run it, it still sees the previous connection. You can do that by clicking on the red box in the console.

### Let’s Recap!

You have accomplished quite a bit so far. You learned how to:

* Create a Spring Security configuration file by adding the configuration, EnableWebSecurity, and HTTPSecurity classes.
* Put your Spring Security filter chain in the Spring Security configuration file using the  configure()  method and adding filters for your HTTP requests.
* Create filters such as  authorizeRequests()  and  antMatchers()  to demonstrate role-based access to different URLs.
* Configure role-based users using the **AuthenticationManagerBuilder** class.
* Create a login form by adding the  loginForm()  filter.
* Set up methods using **RolesAllowed** and  **RequestMapping** classes to create dynamic landing pages for your users.

Looks like you are good to go with your Spring Security authentication page.

In the next part, you'll create a login form using OAuth. OAuth lets you log in with Facebook, Google, and more. To dig deeper into what OAuth is and why we love it, let’s meet back up after a short quiz. 😄

* [#](https://openclassrooms.com/en/courses/5683681-secure-your-web-application-with-spring-security/6695821-configure-role-based-access-control#/id/r-6711896)

## **Identify the Advantages & Applications of OAuth 2.0**

### Identify What OAuth Does

In the last part, you learned about authentication and authorization. We created a simple login form using**Spring Security** to show how to create separate pages for role-based access with an admin and a user. That was your initial demonstration on how to use authentication and authorization in-house.

But creating a secure login is easier said than done. Imagine you’re a web application developer for a small e-commerce company, and you have clients who log on and purchase things using their personal data. A majority of data breaches target small companies because they allocate less funding to handling proper data security measures. Wouldn’t it be helpful if you didn’t have to handle the personal data of the clients? Luckily there is a solution to this problem: **OAuth!**

Now that you’ve rolled that weird acronym off of your tongue, you’re probably wondering what it is:

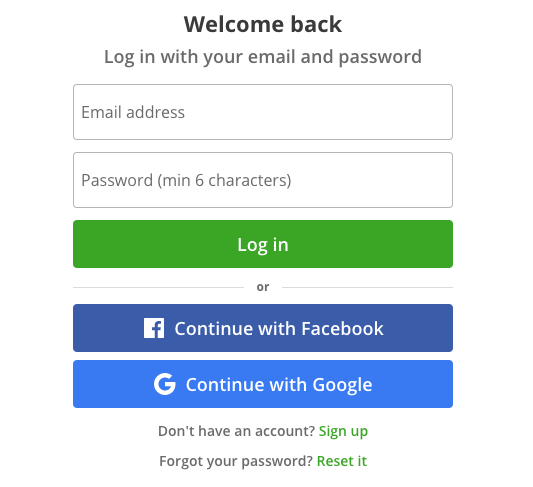
OAuth is an authorization protocol designed to streamline secure login using stateless encrypted tokens to secure user sessions on a web client application.

Huh? 😨

Don’t worry. You'll learn what each of these technical details means as we progress in the chapter. For now, I'll give you a general idea. 😎

In simple terms, OAuth 2.0 allows your clients to create an account on your web application by logging into an account that belongs to an established company like Google, Facebook, Twitter, Okta, or GitHub. These companies have a select server configured for OAuth to make it possible.

At some time, you've probably registered on a site that allows you to create a profile by logging into your Google or a social media account like Twitter or Facebook.

OAuth login page

This web application allows you to create an account with your Google or Facebook credentials. Many people find it easier and quicker to register this way. In this case, Google and Facebook are acting as an**identity provider (IdP).** An identity provider is used to authenticate and manage a user’s identity.

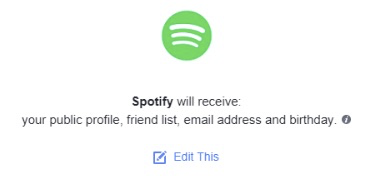
As a security professional, I caution you against connecting your entire life to accounts such as Facebook or Google, but I admit that it is convenient.

This sounds like authentication. Why is OAuth called an authorization protocol?

The established companies (ex. Google, Facebook) handle the authentication part on their servers, as **identity providers.** But when logging on to an app via an identity provider, OAuth delegates **permissions** within that process.

What do you mean by permissions?

Permissions refer to giving access to information. OAuth allows both the user and the server that manages their login (ex. Google) to **authorize what user information the web app is allowed to use.**For example, if you’ve already logged in via an external IdP, you might remember seeing some pop-ups asking you about allowing access to user information:

Authorization request from user with OAuth

In this case, you permit Spotify to access your public profile, your friend list, email address, and birthday.

Facebook handles the login part, and you don’t have to sign up for another account. You don’t even give that website your Facebook login information. As a developer, you don’t have to worry about setting up a secure database to handle everyone’s login because OAuth is taking care of that.

Now that you’ve got an idea of what it is, let’s take a more in-depth look at how OAuth 2.0 works. Once you understand this, it will be much easier to implement into your code using Spring Security.

### Describe the Auth 2.0 Authorization Code Workflow

This authorization protocol has many moving parts, so let’s walk through the actual workflow step by step. There are two types in OAuth 2.0: the**authorization code flow**and the**implicit flow.** The authorization flow is a secure workflow for web applications that allows for customization, which is what we'll be using in this course.

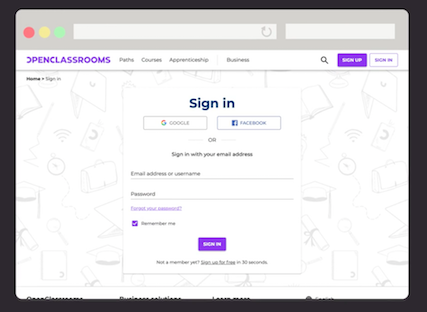
The implicit flow works best for single-page applications (SPA)  and can be described [here](https://auth0.com/docs/flows/concepts/implicit).

##### **Step 1: Connect to Facebook Authorization Server**

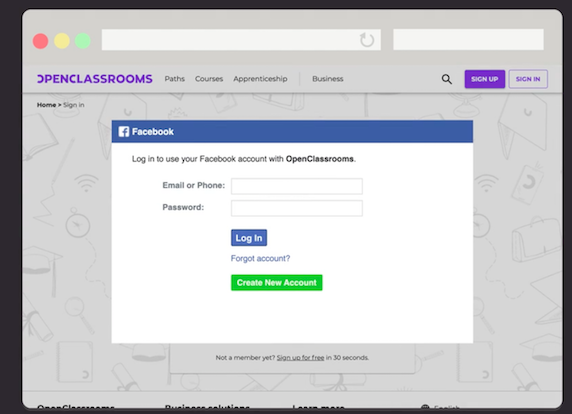
Let’s imagine that you want to login to OpenClassrooms. So far, you’ve encountered two key concepts:

* **User:** That’s you, the person that’s logging into OpenClassrooms with OAuth.
* **Client:** The web application that you are logging into with the ‘Connect to Facebook’ button.

You click to connect to Facebook:

Click the Facebook login option

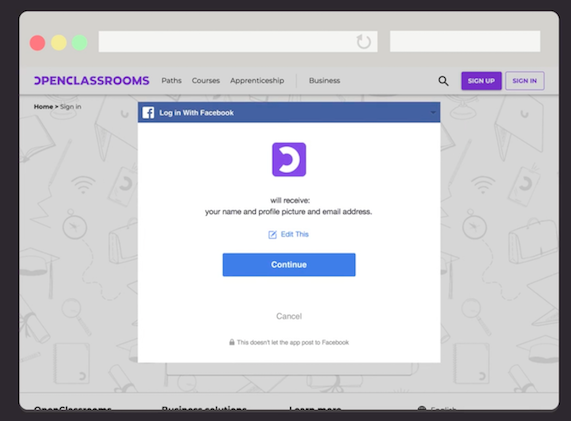
The client sends you to Facebook’s authorization server. The **authorization server** is the Facebook server that you log into with your Facebook credentials. This server registered with OpenClassrooms (which is why you’re allowed to click on the **Log in with Facebook** button and have it work).

Log in via Facebook

You arrive on this page via a redirect URI. When you log into Facebook’s authorization server, you are routed to the Facebook website.  When you are done logging in, Facebook will send you back to OpenClassrooms via a redirectURI.

##### **Step 2: Authorize With OAuth**

Once you log in with your Facebook login credentials, you get a prompt that says, “Will you allow OpenClassrooms access to your contact list?” This is OAuth making sure that you accept the **scope.** Scope refers to personal information that the user authorizes for the client (contacts, name, bio, email address, etc.) - otherwise known as permissions! The specific data requested in the scope is called a**claim.** These claims can be attributes like personal user data, including full name, email address, etc. It can also be specific rights for authorization like read-only or delete access to specific claims like  read-only: email.

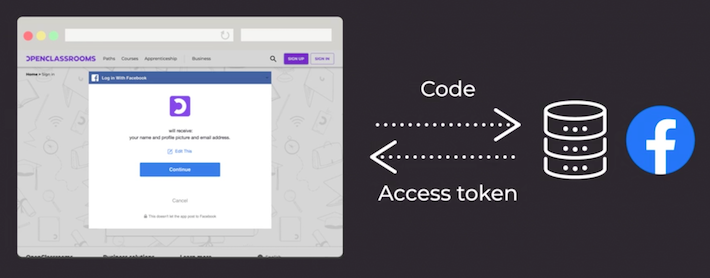
OAuth permissions pop-up

If you answer yes, you will be redirected back to the redirect URI with an **authorization code.** This is a temporary code  that holds your information about your Facebook login.

Although some IdPs like GitHub will generate a redirect URI for you, a predefined redirect URI ensures that your access token does not end up in the wrong place.

##### **Step 3: Send an Authorization Code in Exchange for an Access Token**

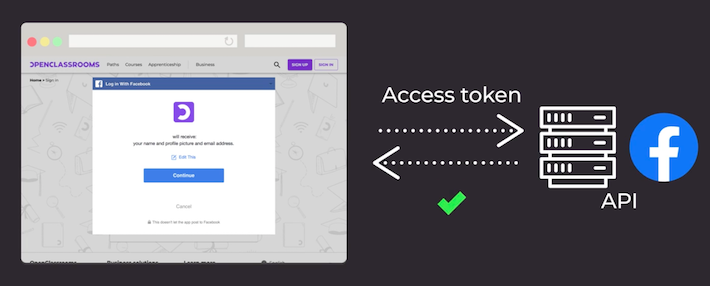
The client sends the authorization code back to Facebook’s authorization server in exchange for an **access token**with your contact list. This token holds authorization information from Facebook.

Interaction with the authorization server

A normal login with cookies is called stateful because the server stores all the information about the user and their session. With tokens, however, the server does not store the user’s credentials and session information making it stateless.

##### Step 4: Validate With the Resource Server

This token is sent to Facebook’s API server, which in this workflow is called a **resource server.** This resource server checks the information on the access token and grants the client app access to the requested user data.

Interaction with the resource server

From now on, every time OpenClassrooms checks to see if you are logged in, it validates you and what information you authorized on Facebook’s resource server.

That’s it.  You now know how OAuth 2.0 works!

### Understand OpenID Connect (OIDC)

While OAuth 2.0 works with external IdPs who provide secure authentication, the OAuth 2.0 process deals primarily with authorization. But what if frenchcheese.crom needed more information like a verified full name and email address?

What's a verified full name and email address?

When you register on specific sites and provide them your email address, your registration is not fully verified until you’ve completed email verification. Usually, you verify once you click on the email verification link, and then you are officially registered. This is one way for an IdP to prove verification of your personal data.

So how does this translate with OAuth 2.0?

OAuth 2.0 is an authorization protocol that returns a **principal** and grant authority object in an access token to validate you after you’ve logged into the IdP and authorized requested personal data. The principal is information about you that the IdP sends back. The **grant authority** is an object that allows your client access to some of your data based on what you authorized when logging into your IdP. It also identifies your role on the client web app.

For the client web app, OAuth 2.0 is good for ***authorization***information but doesn’t ***authenticate*** you on the client web app. Sure, you’ve authenticated on the IdP, but you haven’t authenticated on the client web app, have you?

How can you fix this problem so the client web app has proven and verifiable information to authenticate you? Enter (drumroll, please) this awesome OAuth 2.0 add-on called **OpenID Connect (OIDC). OIDC**can tack on the authentication piece that OAuth 2.0 was not able to do.

So how does it work?

Remember how authentication is essentially proving that you are whom you say you are? Well, with OIDC, the client can get your encrypted personal data from Facebook’s resource server like a verified email, hence **authenticating** you.

This encrypted personal data comes in the form of a **Base64 encrypted ID token** with the access token sent back to the client web app. The ID token holds protected**claim** information, which is the requested user details from the OpenID **scope.**

How is it used? Here are three ways in which OIDC solves the authentication problem by adding security and authentically adding the identity in the external identity provider.

1. Due to Base64 encryption with the IdP’s private key,  
   when authentication requires OIDC, a rogue hacker can’t inject false principal information back to the client because the ID token, access token, and refresh token are sent encrypted via HTTPS.
2. The ID token is short-lived, so authentication is continuously verified. The OAuth 2.0 access token stays valid for 24 hours and can be reinstated with a refresh token, which makes it more challenging to keep up with the validity of a user’s authentication. You can change the access token expiration period [like this](https://auth0.com/docs/dashboard/guides/apis/update-token-lifetime).
3. OIDC uses very specific scopes to work with OAuth 2.0 scope attributes. They are  openid,  profile,  email,  phone, and  address. It works with OAuth 2.0 to return this information in a protected part of the access token with the regular OAuth 2.0 authorization request.

Let’s look at an example where an ID token is used to validate that a user is authenticated.

Your enterprise company uses OAuth 2.0 with **Amazon Web Services (AWS)**as the IdP and **single sign-on (SSO)** to authenticate users to several company **application programming interfaces (APIs).** You can use OpenID Connect for **SSO,** which is used to validate the users on all of the company APIs without having to authenticate several times. The access token is used to validate the users across several APIs and platforms making authentication management secure and simple.

What is an access token vs an ID token?

OAuth uses access tokens because they are more secure than session cookies. Tokens don’t reveal information about the user’s credentials and session information. Access tokens have a username and authorization information. They are not always encrypted or JWT tokens, so buyer beware, they are not guaranteed to be entirely secure.

ID tokens are JWT tokens encrypted with Base64 and an IdP-provided private key. Since the IdP holds the private key, it can only be decoded with the IdP’s permission!

How does adding OIDC change my OAuth workflow?

**Here’s how your OAuth workflow looks when you add OIDC**

The only things that change are Step 1 and Step 3 from the OAuth 2.0 workflow.

* In **Step 1,** you click on “Log in with Facebook,” and behind the scenes of the  OAuth 2.0 workflow with OIDC, the client sends you to the Facebook authorization server with a redirect URI and a scope.  This scope is for OIDC and defines specific claims made by the client application.
* In **Step 3** of the OAuth 2.0 workflow, the **client** sends the **authorization code** back to the Facebook **authorization server** in exchange for the **access token.**  In Step 3 of the OAuth 2.0 workflow with **OIDC,** an **access token** is returned to the **client** with an **ID token.**  This **ID token** contains the requested **claim**information.

Now you know about how OAuth 2.0 authorization works and the benefits of adding OIDC authentication!

### Let’s Recap!

* **OAuth** is an authorization protocol designed to streamline secure login using stateless encrypted tokens to secure user sessions on a web client application.
* OAuth works with **identify providers,** which handle authentication. OAuth handles the authorization of permissions by the user and the identity provider servers.
* **OIDC** can be added to the scope (permissions) so the client can authenticate the user with an ID token and additional verified personal data - called **claims.**

I don’t know about you, but I’m ready to code this into my client web app.  In the next chapter, we will add the default OAuth 2.0 login form using GitHub, and learn to add on OIDC.

[Identify the Advantages & Applications of OAuth 2.0 - Secure Your Web Application With Spring Security - OpenClassrooms](https://openclassrooms.com/en/courses/5683681-secure-your-web-application-with-spring-security/6695836-identify-the-advantages-applications-of-oauth-2-0)

### **Set Up a Login Form to Work With OAuth 2.0 and OIDC**

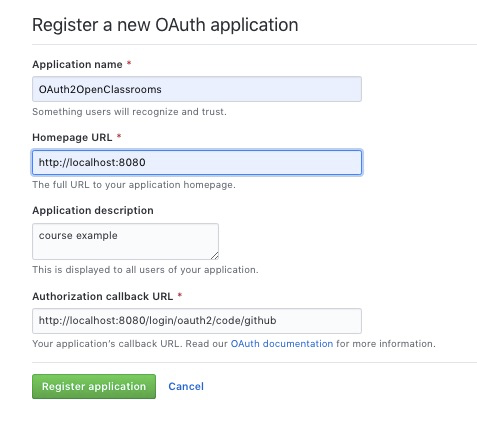
OK, this looks like you’ve got a great, big job ahead of you, but you can do it! You’ve already set up your form with the formLogin() method and in-memory authentication.

You also used **configuration, WebSecurityConfigurerAdapter,**and**HttpSecurity**libraries for your basic Spring Security login form. There won’t be much of a difference in how you start your login form with new security features.

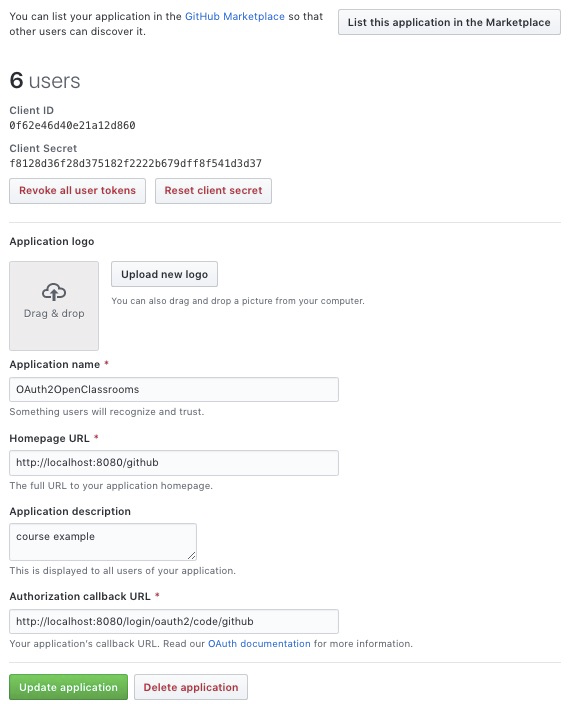
You could use Twitter or Facebook as your authorization servers, but I don’t want to force you to make accounts you might not have. But, if you’re a developer, you’ll likely have or need a **GitHub** account!

To use our client web application with a GitHub login, you will need to register on GitHub to acquire a**client ID**and **client secret.** That’s basically like a username and password for your web app to connect to GitHub's authorization server with OAuth 2.0.

Start by going to the [Github OAuth 2.0 Registration Page](https://github.com/settings/applications/new" \t "_blank) to register your account for an OAuth 2.0 application. On the registration page, put in your localhost address (localhost:8080) instead of an actual domain name. You can also put in a redirect URI, although GitHub generates it's own. This is the URL on your client web app that you are redirected back to after authenticating on GitHub.

GitHub OAuth 2.0 registration page

For example, I’ll put in the default [http://localhost:8080/login/oauth2/code/github<http://localhost:8080/github>](http://localhost:8080/login/oauth2/code/github) for my redirect URI. On success, you will get your own client ID and client secret. Save these for later.

GitHub Auth 2.0 application config

Here are the links to register with [**Google**](https://console.developers.google.com/) and [**Facebook**](https://developers.facebook.com/docs/facebook-login)**.**

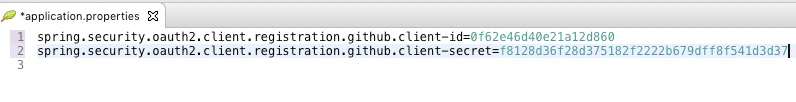
Once you have acquired your**client ID** and**client secret,** you can enter the details in your **application.properties** file in the **src/java/resources** folder.  This will automatically direct your OAuth users to the proper authorization server with your client app.

To use GitHub as your OAuth 2.0 login, you can copy and paste this in your **application.properties** file, but make sure you replace the client ID and client secret with your own registration information provided by GitHub!

spring.security.oauth2.client.registration.github.client-id=<your client ID>

spring.security.oauth2.client.registration.github.client-secret=<client-secret>

All you do is just paste it in so your **application.properties**file should look like this!

GitHub credentials in application.properties

OK, let’s break this application configuration file down.

The client web app sends the client id and client secret to GitHub to log your client web app in and create an OAuth 2.0 authentication form for your client app on GitHub’s authorization server.

#### Configure OAuth 2.0 to the Login Page

Now let’s add OAuth 2.0’s handy default login to your page. All you have to do is add the  oauth2login()  method to your security filter chain in the **SpringSecurityConfig.java**file.

@Override

public void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers("/admin").hasRole("ADMIN")

.antMatchers("/user").hasRole("USER")

.anyRequest().authenticated()

.and()

.formLogin()

.and()

.oauth2Login();

}

Note that I have added two more lines to what we’ve already got in the **SpringSecurityConfig.java** file. This brings up OAuth 2.0’s default login page with a link to the GitHub login page for the web application we registered. Like magic! ✨

I can make you a Spring Security magician T-shirt later, but let’s add a landing page for GitHub to your controller now. Check out the  getGithub()  method for a quick landing page:

@RestController

public class LoginController{

@RolesAllowed("USER")

@RequestMapping("/\*\*")

public String getUser()

{

return "Welcome User";

}

@RolesAllowed({"USER","ADMIN"})

@RequestMapping("/admin")

public String getAdmin()

{

return "Welcome Admin";

}

@RequestMapping("/\*")

public String getGithub()

{

return "Welcome Github user!";

}

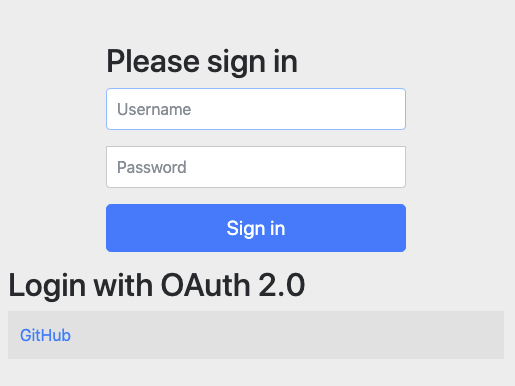
}

Note that I added a super quick mapping for a landing page after authenticating to GitHub. Now it should say Welcome GitHub user! after you authenticate. In the mapping, I put the /\* wildcard URL in the RequestMapping for  getGithub()  because GitHub dynamically generates a redirect URL that corresponds with your Client ID.

#### Login With OAuth 2.0

Here is where you get some real login action going. Build and execute your web application by right-clicking on SpringLoginApplication -> Run as -> Spring Boot App. Now go to your browser and type in [localhost:8080](https://cheezburger.com/).

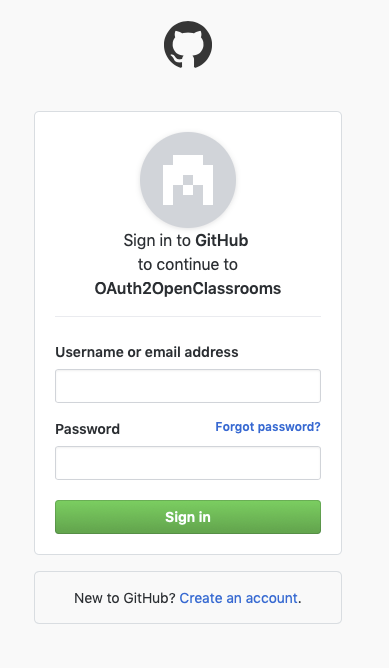
Wow…. this is interesting:

OAuth 2.0 default login form with GitHub

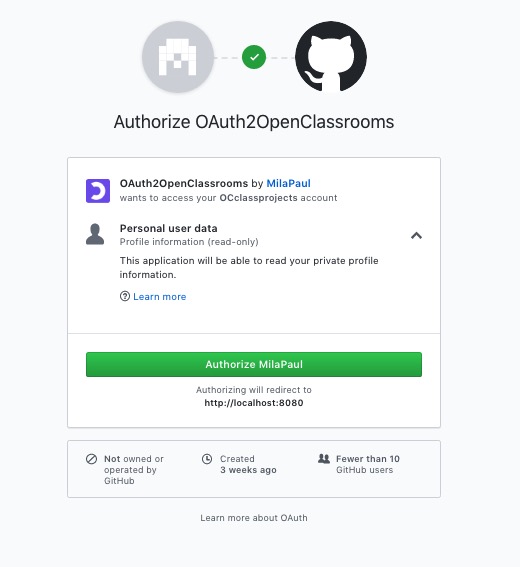
In addition to the Spring Security login form, you have now added OAuth 2.0’s default login form.

Yes!  OAuth 2.0’s login form leaves a lot to be desired, but you can create a custom login with some nice CSS stylesheets that make it look just the way you like it! Check out the [**link**](https://docs.spring.io/spring-security-oauth2-boot/docs/current/reference/htmlsingle/) for ideas on building your own custom login form.

Now **click** on the super plain link to **GitHub’s** authorization server.

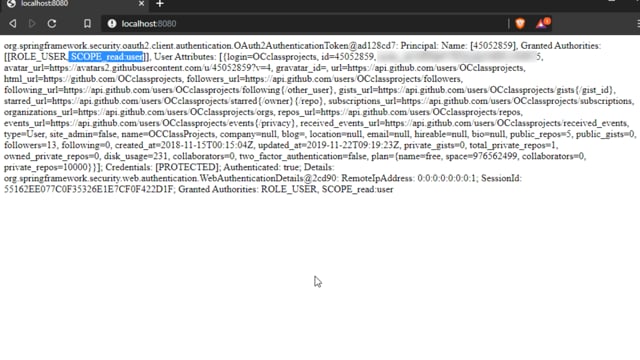
GitHub OAuth 2.0 authentication form

The page will take you to a link specific to your web application. I called mine **OAuth2OpenClassrooms** when I registered the web application on GitHub. Log in with your GitHub credentials.

Authorize personal data from GitHub

You should end up on the landing page. It would be a good idea to give it a personal touch. Check out the next section on how you can do that.

### Explore the Principal User Object

[[](https://openclassrooms.com/en/login?context=courseAccess)](https://openclassrooms.com/en/login?context=courseAccess)

Let’s add the user’s name in your greeting instead of it just saying, “Hello, GitHub user.” How can you do that? Start by looking at the information sent from GitHub and find out what you can extract.

Let’s put in some code that pulls in the information from the **principal.** The principal is an object that is sent over to the client web app and holds various protected and unprotected user details. An example of protected resources would be specific user details and your access token. Unprotected resources hold information like your username so its more readily available. What’s interesting about GitHub, is that unprotected resources hold a lot of information. Let’s check it out!

Let's pull the  Principal user  in as the parameter that holds the  Principal  object in the  getGithub()  method. Typically, you can pull a username from a Principal  using the user.getName() method,  so I'll use that as well.

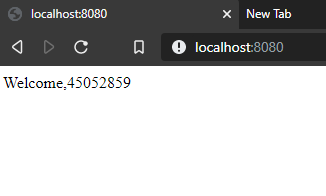
In the end, it should say Welcome, <my Github username>.

public String getGithub(Principal user){

return "Welcome, " + user.getName();

}

It looks like it was a no-go.  It shows some sort of a number:

The user.getName() result from principal user

Well, I guess that doesn't work.  😠 Let's see what kinds of things there are available to print out from the   Principal user. Add a line to the  getGithub()  method to see what was sent over in the Principal user object. You can do that by returning a printout of the  Principal user  object converted to a  String  form using  user.toString().

Do you think a username will show up in the  user.toString()  printout? 😧

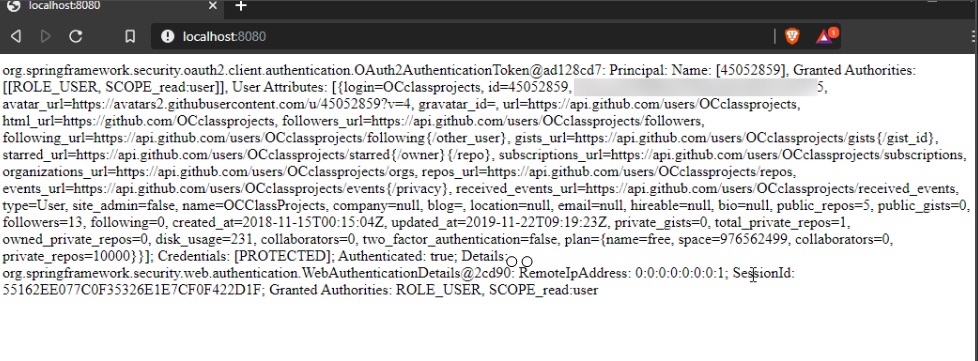
Let's find out.

public String getGithub(Principal user){

return user.toString();

}

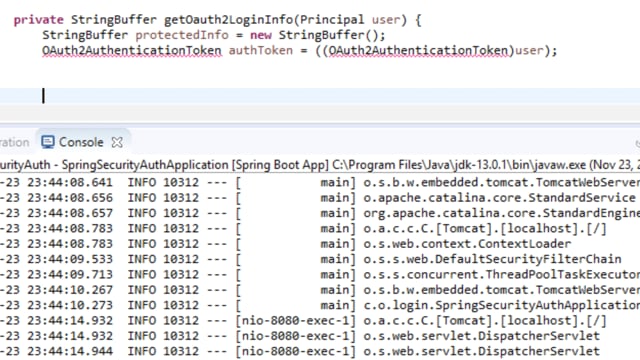
This time the getGithub landing page is completely different. Look at all this information.

GitHub's printout of the principal user object

It looks like the console has all the information sent over from the principal. You can see that something called **grant authorities** has given me a user role, so that looks like the authorization was completed.

It shows the user profile information that I authorized when logging into my GitHub OAuth 2.0 application during authentication. If you remember, that was the **scope.** The scope is read-only profile information you authorize the client web app to have.

### Pull Protected Resources From the Principal User Object

[[](https://openclassrooms.com/en/login?context=courseAccess)](https://openclassrooms.com/en/login?context=courseAccess)

I want to see more important details. I want it to print out my full name and email address. Also, I want to see what my access token looks like, so let’s take a look at that too.

I also want to make this method readily available to use for other **OAuth 2.0 IdPs,** not just GitHub.

First, delete the entire  getGithub()  method in the **BasicController.java** class altogether. You can do that because you’re working with protected resources like the **access token,** and in the next section, the **Base64 ID token.** To get set up to input those protected JWT tokens, the **authentication** class will no longer be useful.

You need a new method for your **LoginController**class to work for your OAuth 2.0 logins. Call the method  getUserInfo() because you want to give it a description that works with **multiple** IdPs, not just GitHub. You still use the**principal** as your parameter as it holds the user details returned from the authorization servers.

@RequestMapping("/\*")

public String getUserInfo(Principal user) {

}

Now you have to figure out a way to input the principal’s different user attributes into memory and output them from memory. Two recommended ways to work with the string content is either with a **HashMap** object or a **StringBuffer**object.

In this example, I use the **StringBuffer** class with an instance called  userinfo  because it's a clean way to create arrays with strings to which I can add more user attributes.

public String getUserInfo(Principal user) {

StringBuffer userInfo= new StringBuffer;

return userInfo.toString();

}

 Let’s create a getter method for your protected user info in the **access token.**This is where you can get your full name, email address, and access token information to add to your  StringBuffer  in the  getUserInfo()  method. Create two more methods to get this information.

These other methods return  StringBuffer  data from the  Principal user object back to the getUserInfo () method.

The **UsernamePasswordAuthenticationToken** is a class that will get the username information after authenticating the token with the user using the  getPrincipal()  method. Then it will be added to the StringBuffer instance,  usernameInfo. You can add this method under the  getUserInfo()  method.

private StringBuffer getUsernamePasswordLoginInfo(Principal user)

{

StringBuffer usernameInfo = new StringBuffer();

UsernamePasswordAuthenticationToken token = ((UsernamePasswordAuthenticationToken) user);

if(token.isAuthenticated()){

User u = (User) token.getPrincipal();

usernameInfo.append("Welcome, " + u.getUsername());

}

else{

usernameInfo.append("NA");

}

return usernameInfo;

}

 The **OAuth2AuthenticationToken** class has methods to use on protected resources such as the protected information in the  user  object. Create a token called authToken of type    **OAuth2AuthenticationToken** and typecast the  Principal  object  user  to a protected token under this class.

private StringBuffer getOauth2LoginInfo(Principal user){

StringBuffer protectedInfo = new StringBuffer();

OAuth2AuthenticationToken authToken = ((OAuth2AuthenticationToken) user);

}

With the **OAuth2AuthenticationToken** class, the authorized client app **(Authorized Client)** has permission to access more protected resources like the access token. For this, the **OAuth2AuthorizedClient** class is used with methods that load authorized client services with the client ID and the user’s username.

But first, create a global private final variable for your **LoginController**class of type **OAuth2AuthenticationToken** and call it  authClientService. Due to its secure nature, it will be set as immutable:

**private final** OAuth2AuthorizedClientService authorizedClientService;

Create a global public constructor for your **BasicController**class:

public LoginController(OAuth2AuthorizedClientService authorizedClientService) {

this.authorizedClientService = authorizedClientService;

}

Assign your **OAuth2AuthorizedClient**object to your authorized client services methods. Notice that the**loadAuthorizedClient**method returns the client matching the client ID, and the principal name in the parameters.

Right now it returns  protectedInfo, which does not contain anything.

private StringBuffer getOauth2LoginInfo(Principal user){

StringBuffer protectedInfo = new StringBuffer();

OAuth2AuthenticationToken authToken = ((OAuth2AuthenticationToken) user);

OAuth2AuthorizedClient authClient = this.authorizedClientService.loadAuthorizedClient(authToken.getAuthorizedClientRegistrationId(), authToken.getName());

return protectedInfo;

}

 Take the StringBuffer information pulled from the principal user object with **UsernamePasswordAuthenticationToken** and **OAuth2AuthenticationToken,** and append  the returned  usernameInfo  and protectedInfo,  to  userInfo in your  getUserInfo() method.

public String getUserInfo(Principal user) {

StringBuffer userInfo= new StringBuffer;

if(user instanceof UsernamePasswordAuthenticationToken){

userInfo.append(getUsernamePasswordLoginInfo(user));

}

else if(user instanceof OAuth2AuthenticationToken){

userInfo.append(getOauth2LoginInfo(user));

}

return userInfo.toString();

}

Now you have the information you need to print out the full name, email address, and access token. Use a  **HashMap**variable called  userAttributes  for *<key, value>* pairs that match the attributes (the user details) with the key (title).

Go back to your  getOauth2LoginInfo()  method, and print out the information for the user when they log on. The name and email are attributes that are pulled from the **principal.** Now there’s the access token.  Add it using the getAccessToken() and getTokenValue() methods and print that out too. The userToken variable is assigned the **access token** value, so it will contain the encrypted character string that is the access token. This is all appended onto  protectedInfo, and returned to its function call in get  UserInfo().

Let's see what happens when you authenticate with the Google login.

private StringBuffer getOauth2LoginInfo(Principal user){

StringBuffer protectedInfo = new StringBuffer();

OAuth2AuthenticationToken authToken = ((OAuth2AuthenticationToken) user);

OAuth2AuthorizedClient authClient = this.authorizedClientService.loadAuthorizedClient(authToken.getAuthorizedClientRegistrationId(), authToken.getName());

if(authToken.isAuthenticated()){

Map<String,Object> userAttributes = ((DefaultOAuth2User) authToken.getPrincipal()).getAttributes();

String userToken = authClient.getAccessToken().getTokenValue();

protectedInfo.append("Welcome, " + userAttributes.get("name")+"<br><br>");

protectedInfo.append("e-mail: " + userAttributes.get("email")+"<br><br>");

protectedInfo.append("Access Token: " + userToken+"<br><br>");

}

else{

protectedInfo.append("NA");

}

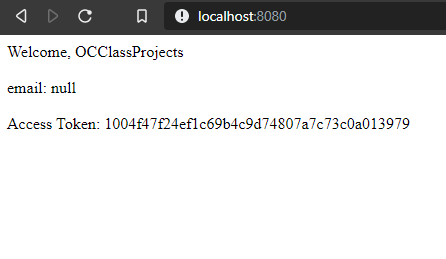
return protectedInfo;

}

Will I be able to see this information?

You definitely don’t want to print out someone’s access token for the world to see on your client app to make a hacker’s life easier (it’s a protected token!), but I'm adding the access token in our print out on the welcome page FOR DEMONSTRATION PURPOSES ONLY!!

Here is a screenshot of the results that I got.

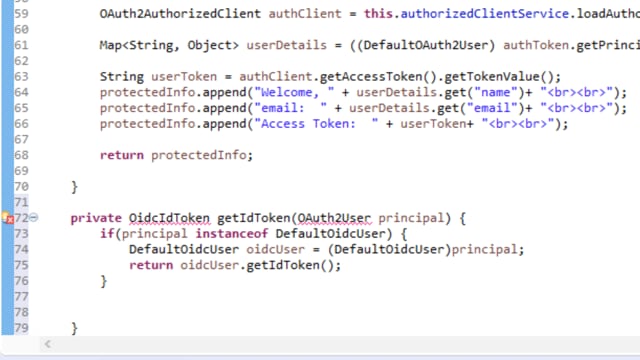
Pulling the username, email and access token from GitHub

So why doesn't the email address show? 😭

When you pull information from  Principal user  object, you have unprotected and protected information in there. You can use some classes like **principal** and **authentication** to pull unprotected information. Classes that can pull protected information have a layer of abstraction and require more validation. GitHub puts email address information under the protected layer, so it is likely that the only way you can pull it is from the ID Token with an  openid   email  claim in the**scope.**

This leads me to OpenID Connect. Let's go to the next section to check out how we can pull protected information from an ID Token by acquiring and decoding it!

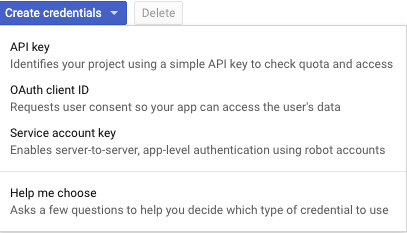
### Use Open ID Connect With Google

[[](https://openclassrooms.com/en/login?context=courseAccess)](https://openclassrooms.com/en/login?context=courseAccess)

You learned a lot about OAuth 2.0 and how you can get some client information with the principal and the user details. But to tell you the truth, not all IdPs send over as much detailed information about a user as GitHub.

Let’s switch gears and talk about how it’s so essential to use OpenID Connect for authentication. Google is an excellent resource for OAuth 2.0, and it has excellent support for OpenID Connect.

It won’t take long to add Google to your OAuth 2.0 login form. First, go to the [Google OAuth 2.0 Credential Registration](https://console.developers.google.com/apis/credentials) and click on Credentials in the left pane -> Create  Credentials -> OAuth client ID.

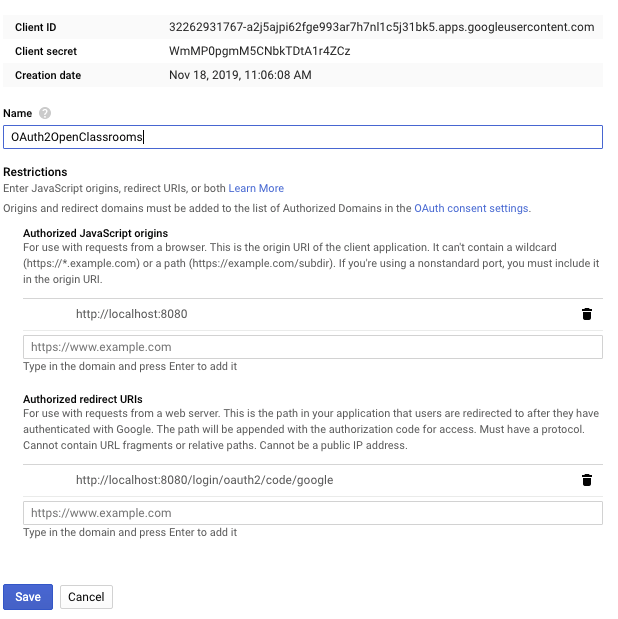
Google OAuth 2.0 credential registration

In the form, create a name for your OAuth 2.0 Client web application. To work on your  localhost:8080  web server, set up your configuration like this:

**Restrictions:**

Javascript Origins: [http://localhost:8080](http://localhost:8080/)

Authorized redirect URI: http://localhost:8080/login/oauth2/code/google

Google OAuth2 credentials

Save your Client ID and Client secret so you can put Google properties in your **application.properties** folder under your GitHub properties.

spring.security.oauth2.client.registration.google.client-id=<clientid>.apps.googleusercontent.com

spring.security.oauth2.client.registration.google.client-secret=<clientsecret>

spring.security.oauth2.client.registration.google.scope=openid,profile,email

 The key to adding OpenID Connect to your initial request to the login server is the **scope** for  openid  and the **claims.** I added  openid  as the scope with  profile  and  email  as the claims. These claims should come back with the **ID token** and the **access token** in the  Principal user  object.

Now you have to add your ID token retrieval to your **LoginController.java** file. You do this in order for the  openid  scope to work.

Luckily, you can append some code to get your ID token in your getOAuth2LoginInfo() method.  We have managed to pull in the  Principal user  details and access token. Now you have to get your ID token that holds your claim information.

The **ID token** is a protected resource, so add to your get  Oauth2LoginInfo()  method to retrieve it. Use the OAuth2User class to create a variable principal for your protected OAuth2AuthenticatedToken typecasted user object to retrieve the ID token and print out the encrypted character string. The getPrincipal() method gets all the information the OIDC classes need to make a new ID token.

OAuth2User principal = ((OAuth2AuthenticationToken) user).getPrincipal();

First, create a new getter method for your ID token. Return an ID token of type **OidcIdToken,** a class used to make ID tokens. DefaultOidcUser is a specific subset of OidcUser and ultimately, OAuth2User. By checking to see if  principal  is of this subset in the if statement, a variable  called  oidcUser  of type DefaultOidcUser is created using the required information provided from the protected principal information of the overarching OAuth2User class.

private OidcIdToken getIdToken(OAuth2User principal){

if(principal instanceof DefaultOidcUser) {

DefaultOidcUser oidcUser = (DefaultOidcUser)principal;

return oidcUser.getIdToken();

}

return null;

}

Now that you’ve sent your ID token back to your  getOauth2LoginInfo()  method, add some code so it knows what to do with it. Add the code directly below the following append statement in your  getOauth2LoginInfo()  method:

protectedInfo.append("Access Token: " + userToken+"<br><br>");

Start with the **OidcIdToken**class to create a variable  idToken  to hold your ID token. It gets assigned to the function call to getIdToken() that sends in the principal as a parameter.

OidcIdToken idToken = getIdToken(principal);

Now append the ID token character string to  protectedInfo. The ID token holds the claims info that you requested in your scope. If you remember, you created your scope for Google with the values  openid   profile   email. With  openid  as the required default, your additional claims were for verified profile and email information to authenticate your user.

In the code below, you have a **HashMap** for your claims which you retrieve using the  getClaims()  method specified in the **DefaultOidcUser**class:

if(idToken != null) {

protectedInfo.append("idToken value: " + idToken.getTokenValue()+"<br><br>");

protectedInfo.append("Token mapped values <br><br>");

Map<String, Object> claims = idToken.getClaims();

for (String key : claims.keySet()) {

protectedInfo.append(" " + key + ": " + claims.get(key)+"<br>");

}

}

When you use OIDC, you can specify the specific information you want the user to authorize in the scope. This specific information is called a **claim,** and there is a list of authorization claims [**here**](https://auth0.com/docs/scopes/current/oidc-scopes)**.**

You have gotten to the heart of securing your login page with OAuth 2.0! I hope you enjoyed it because there are limitless opportunities in how you can set it up securely using token-based authorization and authentication.

Also, now that you’ve gotten through the lesson, you can add OAuth 2.0 and OpenID Connect as a skillset to your resume because you are now a Spring Security/OAuth 2.0/OpenID Connect ninja!

### Let’s Recap!

You have gone through the following steps to create a default OAuth 2.0 login with OIDC:

1. Configure the **OAuth 2.0 default login** in the security filter chain.
2. Register your web application with GitHub and Google for a**client ID** and **client secret.**
3. Add the **GitHub configuration** to the application.properties file.
4. **Extract principal user details** and **access token** from GitHub.
5. Configure **OIDC** with GitHub in the application.properties file.
6. **Extract principal, access token,** and **ID token** from Google.
7. **Decode the ID token** to extract claim information.

Now there’s more. You’ve added some great options to authorize users to your web application, but how can you be sure that regular users can’t get on pages they aren’t authorized to see?

That’s important, so we should probably set up a test method to make sure you can stay on top of that! Let’s explore how you can create that method in the next chapter.

* [#](https://openclassrooms.com/en/courses/5683681-secure-your-web-application-with-spring-security/6695831-configure-oauth-2-0-with-openid-connect-on-a-spring-web-application#/id/r-6756408)