* [**PROJECTS**](https://spring.io/projects)
* [**GUIDES**](https://spring.io/guides)
* [**BLOG**](https://spring.io/blog)

Top of Form



Bottom of Form

TUTORIAL

**Spring Boot and OAuth2**

This guide shows you how to build a sample app doing various things with "social login" using [OAuth2](https://tools.ietf.org/html/rfc6749) and [Spring Boot](https://projects.spring.io/spring-boot/). It starts with a simple, single-provider single-sign on, and works up to a self-hosted OAuth2 Authorization Server with a choice of authentication providers ([Facebook](https://developers.facebook.com/)or [Github](https://developer.github.com/)). The samples are all single-page apps using Spring Boot and Spring OAuth on the back end. They also all use plain [jQuery](https://jquery.org/) on the front end, but the changes needed to convert to a different JavaScript framework or to use server side rendering would be minimal.

Because one of the samples is a full OAuth2 Authorization Server we have used the [shim JAR](https://docs.spring.io/spring-security-oauth2-boot/docs/current/reference/htmlsingle/)which supports bridging from Spring Boot 2.0 to the old Spring Security OAuth2 library. The simpler samples could also be implemented using the native OAuth2 support in [Spring Boot](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#boot-features-security-oauth2)security features. The configuration is very similar.

There are several samples building on each other adding new features:

* [**simple**](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_simple): a very basic static app with just a home page and unconditional login through via Spring Boot’s @EnableOAuth2Sso (if you visit the home page you will be automatically redirected to Facebook).
* [**click**](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_click): adds an explicit link that the user has to click to login.
* [**logout**](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_logout): adds a logout link as well for authenticated users.
* [**manual**](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_manual): shows how the @EnableOAuth2Sso works by unpicking it and configuring all its pieces manually.
* [**github**](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_github): adds a second login provider in Github, so the user can choose on the home page which one to use.
* [**auth-server**](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_authserver): turns the app into a fully-fledged OAuth2 Authorization Server, able to issue its own tokens, but still using the external OAuth2 providers for authentication.
* [**custom-error**](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_custom_error): adds an error message for unauthenticated users, and a custom authentication based on Github API.

|  |  |
| --- | --- |
|  | The changes needed to migrate from one app to the next one in the feature ladder can be tracked in the source (the source code is [in Github](https://github.com/spring-guides/tut-spring-boot-oauth2)). The first 6 changes in the repository are transforming a single app so you can easily see the differences. Any further differences you might see between the app in those early commits and the final state you see in the guide are cosmetic. |

Each of them can be imported into an IDE and there is a main class SocialApplication that you can run there to start the apps. They all come up with a home page on [http://localhost:8080](http://localhost:8080/) (and all require that you have at least a Facebook account if you want to log in and see the content). You can also run all the apps on the command line using mvn spring-boot:run or by building the jar file and running it with mvn package and java -jar target/\*.jar (per the [Spring Boot docs](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#getting-started-first-application-run) and other [available documentation](https://spring.io/guides/gs/spring-boot/)). There is no need to install Maven if you use the [wrapper](https://github.com/takari/maven-wrapper) at the top level, e.g.

$ cd simple

$ ../mvnw package

$ java -jar target/\*.jar

|  |  |
| --- | --- |
|  | The apps all work on localhost:8080 because they use OAuth2 clients registered with Facebook and Github for that address. To run them on a different host or port, you need to register your own apps and put the credentials in the config files. There is no danger of leaking your Facebook or Github credentials beyond localhost if you use the default values, but be careful what you expose on the internet, and don’t put your own app registrations in public source control. |

**Single Sign On With Facebook**

In this section we create a minimal application that uses Facebook for authentication. This will be quite easy if we take advantage of the autoconfiguration features in Spring Boot.

**Creating a New Project**

First we need to create a Spring Boot application, which can be done in a number of ways. The easiest is to go to [http://start.spring.io](https://start.spring.io/) and generate an empty project (choosing the "Web" dependency as starting points). Equivalently do this on the command line:

$ mkdir ui && cd ui

$ curl https://start.spring.io/starter.tgz -d style=web -d name=simple | tar -xzvf -

You can then import that project into your favourite IDE (it’s a normal Maven Java project by default), or just work with the files and "mvn" on the command line.

**Add a Home Page**

In your new project create an index.html in the "src/main/resources/static" folder. You should add some style sheets and java script links so the result looks like this:

index.html

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8"/>

<meta http-equiv="X-UA-Compatible" content="IE=edge"/>

<title>Demo</title>

<meta name="description" content=""/>

<meta name="viewport" content="width=device-width"/>

<base href="/"/>

<link rel="stylesheet" type="text/css" href="/webjars/bootstrap/css/bootstrap.min.css"/>

<script type="text/javascript" src="/webjars/jquery/jquery.min.js"></script>

<script type="text/javascript" src="/webjars/bootstrap/js/bootstrap.min.js"></script>

</head>

<body>

<h1>Demo</h1>

<div class="container"></div>

</body>

</html>

None of this is necessary to demonstrate the OAuth2 login features, but we want to have a nice looking UI in the end, so we might as well start with some basic stuff in the home page.

If you start the app and load the home page you will notice that the stylesheets have not been loaded. So we need to add those as well, and we can do that by adding some dependencies:

pom.xml

<dependency>

<groupId>org.webjars</groupId>

<artifactId>jquery</artifactId>

<version>2.1.1</version>

</dependency>

<dependency>

<groupId>org.webjars</groupId>

<artifactId>bootstrap</artifactId>

<version>3.2.0</version>

</dependency>

<dependency>

<groupId>org.webjars</groupId>

<artifactId>webjars-locator-core</artifactId>

</dependency>

We added Twitter bootstrap and jQuery (which is all we need right now). The other dependency is the webjars "locator" which is provided as a library by the webjars site, and which can be used by Spring to locate static assets in webjars without needing to know the exact versions (hence the versionless /webjars/\*\* links in the index.html). The webjar locator is activated by default in a Spring Boot app as long as you don’t switch off the MVC autoconfiguration.

With those changes in place we should have a nice looking home page for our app.

**Securing the Application**

To make the application secure we just need to add Spring Security as a dependency. If we do that the default will be to secure it with HTTP Basic, so since we want to do a "social" login (delegate to Facebook), we add the Spring Security OAuth2 dependency as well:

pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.security.oauth.boot</groupId>

<artifactId>spring-security-oauth2-autoconfigure</artifactId>

<version>2.0.0.RELEASE</version>

</dependency>

To make the link to Facebook we need an @EnableOAuth2Sso annotation on our main class:

SocialApplication.java

@SpringBootApplication

@EnableOAuth2Sso

public class SocialApplication {

...

}

and some configuration (converting application.properties to YAML for better readability):

application.yml

security:

oauth2:

client:

clientId: 233668646673605

clientSecret: 33b17e044ee6a4fa383f46ec6e28ea1d

accessTokenUri: https://graph.facebook.com/oauth/access\_token

userAuthorizationUri: https://www.facebook.com/dialog/oauth

tokenName: oauth\_token

authenticationScheme: query

clientAuthenticationScheme: form

resource:

userInfoUri: https://graph.facebook.com/me

...

The configuration refers to a client app registered with Facebook in their [developers](https://developers.facebook.com/) site, in which you have to supply a registered redirect (home page) for the app. This one is registered to "localhost:8080" so it only works in an app running on that address.

With that change you can run the app again and visit the home page at [http://localhost:8080](http://localhost:8080/). Instead of the home page you should be redirected to login with Facebook. If you do that, and accept any authorizations you are asked to make, you will be redirected back to the local app and the home page will be visible. If you stay logged into Facebook, you won’t have to re-authenticate with this local app, even if you open it in a fresh browser with no cookies and no cached data. (That’s what Single Sign On means.)

|  |  |
| --- | --- |
|  | if you are working through this section with the sample application, be sure to clear your browser cache of cookies and HTTP Basic credentials. In Chrome the best way to do that for a single server is to open a new incognito window. |

It is safe to grant access to this sample because only the app running locally can use the tokens and the scope it asks for is limited. Be aware of what you are approving when you log into apps like this though: they might ask for permission to do more than you are comfortable with (e.g. they might ask for permission to change your personal data, which would be unlikely to be in your interest).

**What Just Happened?**

The app you just wrote, in OAuth2 terms, is a Client Application and it uses the [authorization code grant](https://tools.ietf.org/html/rfc6749#section-4) to obtain an access token from Facebook (the Authorization Server). It then uses the access token to ask Facebook for some personal details (only what you permitted it to do), including your login ID and your name. In this phase facebook is acting as a Resource Server, decoding the token that you send and checking it gives the app permission to access the user’s details. If that process is successful the app inserts the user details into the Spring Security context so that you are authenticated.

If you look in the browser tools (F12 on Chrome) and follow the network traffic for all the hops, you will see the redirects back and forth with Facebook, and finally you land back on the home page with a new Set-Cookie header. This cookie (JSESSIONID by default) is a token for your authentication details for Spring (or any servlet-based) applications.

So we have a secure application, in the sense that to see any content a user has to authenticate with an external provider (Facebook). We wouldn’t want to use that for an internet banking website, but for basic identification purposes, and to segregate content between different users of your site, it’s an excellent starting point, which explains why this kind of authentication is very popular these days. In the next section we are going to add some basic features to the application, and also make it a bit more obvious to users what is going on when they get that initial redirect to Facebook.

**Add a Welcome Page**

In this section we modify the [simple](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_simple) app we just built, by adding an explicit link to login with Facebook. Instead of being redirected immediately, the new link will be visible on the home page, and the user can choose to login or to stay unauthenticated. Only when the user has clicked on the link will he be shown the secure content.

**Conditional Content in Home Page**

To render some content conditional on whether the user is authenticated or not we could use server side rendering (e.g. with Freemarker or Tymeleaf), or we can just ask the browser to to it, using some JavaScript. To do that we are going to use [AngularJS](https://angularjs.org/), but if you prefer to use a different framework, it shouldn’t be very hard to translate the client code.

To get started with the dynamic content we need to mark up the HTML in parts of it are going to display it:

index.html

<div class="container unauthenticated">

With Facebook: <a href="/login">click here</a>

</div>

<div class="container authenticated" style="display:none">

Logged in as: <span id="user"></span>

</div>

This HTML sets us up with a need for some client side code that manipulates the authenticated, unauthenticated and user elements. Here’s a simple implementation of those features (drop them in at the end of the <body>):

index.html

<script type="text/javascript">

$.get("/user", function(data) {

$("#user").html(data.userAuthentication.details.name);

$(".unauthenticated").hide()

$(".authenticated").show()

});

</script>

**Server Side Changes**

For this to work we need some changes on the server side. The "home" controller needs an endpoint at "/user" that describes the currently authenticated user. That’s quite easy to do, e.g. in our main class:

SocialApplication

@SpringBootApplication

@EnableOAuth2Sso

@RestController

public class SocialApplication {

@RequestMapping("/user")

public Principal user(Principal principal) {

return principal;

}

public static void main(String[] args) {

SpringApplication.run(SocialApplication.class, args);

}

}

Note the use of @RestController and @RequestMapping and the java.security.Principalwe inject into the handler method.

|  |  |
| --- | --- |
|  | It’s not a great idea to return a whole Principal in a /user endpoint like that (it might contain information you would rather not reveal to a browser client). We only did it to get something working quickly. Later in the guide we will convert the endpoint to hide the information we don’t need the browser to have. |

This app will now work fine and authenticate as before, but without giving the user a chance to click on the link we just provided. To make the link visible we also need to switch off the security on the home page by adding a WebSecurityConfigurer:

SocialApplication

@SpringBootApplication

@EnableOAuth2Sso

@RestController

public class SocialApplication extends WebSecurityConfigurerAdapter {

...

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.antMatcher("/\*\*")

.authorizeRequests()

.antMatchers("/", "/login\*\*", "/webjars/\*\*", "/error\*\*")

.permitAll()

.anyRequest()

.authenticated();

}

}

Spring Boot attaches a special meaning to a WebSecurityConfigurer on the class that carries the @EnableOAuth2Sso annotation: it uses it to configure the security filter chain that carries the OAuth2 authentication processor. So all we need to do to make our home page visible is to explicitly authorizeRequests() to the home page and the static resources it contains (we also include access to the login endpoints which handle the authentication). All other requests (e.g. to the /user endpoint) require authentication.

|  |  |
| --- | --- |
|  | /error\*\* is an unprotected path because we want Spring Boot to bew able to render errors if there is a problem in the app, even if the user is unauthenticated. |

With that change in place the application is complete, and if you run it and visit the home page you should see a nicely styled HTML link to "login with Facebook". The link takes you not directly to Facebook, but to the local path that processes the authentication (and sends a redirect to Facebook). Once you have authenticated you get redirected back to the local app, where it now displays your name (assuming you have set up your permissions in Facebook to allow access to that data).

**Add a Logout Button**

In this section we modify the [click](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_click) app we built by adding a button that allows the user to log out of the app. This seems like a simple feature, but it requires a bit of care to implement, so it’s worth spending some time discussing exactly how to do it. Most of the changes are to do with the fact that we are transforming the app from a read-only resource to a read-write one (logging out requires a state change), so the same changes would be needed in any realistic application that wasn’t just static content.

**Client Side Changes**

On the client we just need to provide a logout button and some JavaScript to call back to the server to ask for the authentication to be cancelled. First, in the "authenticated" section of the UI, we add the button:

index.html

<div class="container authenticated">

Logged in as: <span id="user"></span>

<div>

<button onClick="logout()" class="btn btn-primary">Logout</button>

</div>

</div>

and then we provide the logout() function that it refers to in the JavaScript:

index.html

var logout = function() {

$.post("/logout", function() {

$("#user").html('');

$(".unauthenticated").show();

$(".authenticated").hide();

})

return true;

}

The logout() function does a POST to /logout and then clears the dynamic content. Now we can switch over to the server side to implement that endpoint.

**Adding a Logout Endpoint**

Spring Security has built in support for a /logout endpoint which will do the right thing for us (clear the session and invalidate the cookie). To configure the endpoint we simply extend the existing configure() method in our WebSecurityConfigurer:

SocialApplication.java

@Override

protected void configure(HttpSecurity http) throws Exception {

http.antMatcher("/\*\*")

... // existing code here

.and().logout().logoutSuccessUrl("/").permitAll();

}

The /logout endpoint requires us to POST to it, and to protect the user from Cross Site Request Forgery (CSRF, pronounced "sea surf"), it requires a token to be included in the request. The value of the token is linked to the current session, which is what provides the protection, so we need a way to get that data into our JavaScript app.

Many JavaScript frameworks have built in support for CSRF (e.g. in Angular they call it XSRF), but it is often implemented in a slightly different way than the out-of-the box behaviour of Spring Security. For instance in Angular the front end would like the server to send it a cookie called "XSRF-TOKEN" and if it sees that, it will send the value back as a header named "X-XSRF-TOKEN". We can implement the same behaviour with our simple jQuery client, and then the server side changes will work with other front end implementations with no or very few changes. To teach Spring Security about this we need to add a filter that creates the cookie and also we need to tell the existing CRSF filter about the header name. In the WebSecurityConfigurer:

SocialApplication.java

@Override

protected void configure(HttpSecurity http) throws Exception {

http.antMatcher("/\*\*")

... // existing code here

.and().csrf().csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());

}

**Adding the CSRF Token in the Client**

Since we are not using a higher level framework in this sample, we need to explicitly add the CSRF token, which we made available as a cookie from the backend. To make the code a bit simpler, we include an additional library:

pom.xml

<dependency>

<groupId>org.webjars</groupId>

<artifactId>js-cookie</artifactId>

<version>2.1.0</version>

</dependency>

Import it in HTML:

index.html

<script type="text/javascript" src="/webjars/js-cookie/js.cookie.js"></script>

then we can use Cookies convenience methods in xhr:

index.html

$.ajaxSetup({

beforeSend : function(xhr, settings) {

if (settings.type == 'POST' || settings.type == 'PUT'

|| settings.type == 'DELETE') {

if (!(/^http:.\*/.test(settings.url) || /^https:.\*/

.test(settings.url))) {

// Only send the token to relative URLs i.e. locally.

xhr.setRequestHeader("X-XSRF-TOKEN",

Cookies.get('XSRF-TOKEN'));

}

}

}

});

**Ready To Roll!**

With those changes in place we are ready to run the app and try out the new logout button. Start the app and load the home page in a new browser window. Click on the "login" link to take you to Facebook (if you are already logged in there you might not notice the redirect). Click on the "Logout" button to cancel the current session and return the app to the unauthenticated state. If you are curious you should be able to see the new cookies and headers in the requests that the browser exchanges with the local server.

Remember that now the logout endpoint is working with the browser client, then all other HTTP requests (POST, PUT, DELETE, etc.) will also work just as well. So this should be a good platform for an application with some more realistic features.

**Manual Configuration of OAuth2 Client**

In this section we modify the [logout](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_logout) app we already built by picking apart the "magic" in the @EnableOAuth2Sso annotation, manually configuring everything in there to make it explicit.

**Clients and Authentication**

There are 2 features behind @EnableOAuth2Sso: the OAuth2 client, and the authentication. The client is re-usable, so you can also use it to interact with the OAuth2 resources that your Authorization Server (in this case Facebook) provides (in this case the [Graph API](https://developers.facebook.com/docs/graph-api)). The authentication piece aligns your app with the rest of Spring Security, so once the dance with Facebook is over your app behaves exactly like any other secure Spring app.

The client piece is provided by Spring Security OAuth2 and switched on by a different annotation @EnableOAuth2Client. So the first step in this transformation is to remove the @EnableOAuth2Sso and replace it with the lower level annotation:

SocialApplication

@SpringBootApplication

@EnableOAuth2Client

@RestController

public class SocialApplication extends WebSecurityConfigurerAdapter {

...

}

Once that is done we have some stuff created for us that will be useful. First off we can inject an OAuth2ClientContext and use it to build an authentication filter that we add to our security configuration:

SocialApplication

@SpringBootApplication

@EnableOAuth2Client

@RestController

public class SocialApplication extends WebSecurityConfigurerAdapter {

@Autowired

OAuth2ClientContext oauth2ClientContext;

@Override

protected void configure(HttpSecurity http) throws Exception {

http.antMatcher("/\*\*")

...

.and().addFilterBefore(ssoFilter(), BasicAuthenticationFilter.class);

}

...

}

this filter is created in new method where we use the OAuth2ClientContext:

SocialApplication

private Filter ssoFilter() {

OAuth2ClientAuthenticationProcessingFilter facebookFilter = new OAuth2ClientAuthenticationProcessingFilter("/login/facebook");

OAuth2RestTemplate facebookTemplate = new OAuth2RestTemplate(facebook(), oauth2ClientContext);

facebookFilter.setRestTemplate(facebookTemplate);

UserInfoTokenServices tokenServices = new UserInfoTokenServices(facebookResource().getUserInfoUri(), facebook().getClientId());

tokenServices.setRestTemplate(facebookTemplate);

facebookFilter.setTokenServices(tokenServices);

return facebookFilter;

}

the filter also needs to know about the client registration with Facebook:

SocialApplication

@Bean

@ConfigurationProperties("facebook.client")

public AuthorizationCodeResourceDetails facebook() {

return new AuthorizationCodeResourceDetails();

}

and to complete the authentication it needs to know where the user info endpoint is in Facebook:

SocialApplication

@Bean

@ConfigurationProperties("facebook.resource")

public ResourceServerProperties facebookResource() {

return new ResourceServerProperties();

}

Note that with both these "static" data objects (facebook() and facebookResource()) we used a @Bean decorated as @ConfigurationProperties. That means that we can convert the application.yml to a slightly new format, where the prefix for configuration is facebookinstead of security.oauth2:

application.yml

facebook:

client:

clientId: 233668646673605

clientSecret: 33b17e044ee6a4fa383f46ec6e28ea1d

accessTokenUri: https://graph.facebook.com/oauth/access\_token

userAuthorizationUri: https://www.facebook.com/dialog/oauth

tokenName: oauth\_token

authenticationScheme: query

clientAuthenticationScheme: form

resource:

userInfoUri: https://graph.facebook.com/me

Finally, we changed the path to the login to be facebook-specific in the Filter declaration above, so we need to make the same change in the HTML:

index.html

<h1>Login</h1>

<div class="container unauthenticated">

<div>

With Facebook: <a href="/login/facebook">click here</a>

</div>

</div>

**Handling the Redirects**

The last change we need to make is to explicitly support the redirects from our app to Facebook. This is handled in Spring OAuth2 with a servlet Filter, and the filter is already available in the application context because we used @EnableOAuth2Client. All that is needed is to wire the filter up so that it gets called in the right order in our Spring Boot application. To do that we need a FilterRegistrationBean:

SocialApplication.java

@Bean

public FilterRegistrationBean oauth2ClientFilterRegistration(

OAuth2ClientContextFilter filter) {

FilterRegistrationBean registration = new FilterRegistrationBean();

registration.setFilter(filter);

registration.setOrder(-100);

return registration;

}

We autowire the already available filter, and register it with a sufficiently low order that it comes **before** the main Spring Security filter. In this way we can use it to handle redirects signaled by expceptions in authentication requests.

With these changes in place the app is good to go, and at runtime is equivalent to the [logout](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_logout)sample we built in the last section. Breaking the configuration down and making it explicit teaches us that there is nothing magical about what Spring Boot is doing (it’s just configuration boiler plate), and it also prepares our application for extending the features provided automatically out of the box, adding our own opinions and business requirements.

**Login with Github**

In this section we modify the [app](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_manual) we built already, adding a link so the user can choose to authenticate with Github, in addition to the original link to Facebook.

**Adding the Github Link**

In the client the change is trivial, we just add another link:

index.html

<div class="container unauthenticated">

<div>

With Facebook: <a href="/login/facebook">click here</a>

</div>

<div>

With Github: <a href="/login/github">click here</a>

</div>

</div>

In principle, once we start adding authentication providers, we may need to be more careful about the data returned from the "/user" endpoint. It turns out that Github and Facebook both have a "name" field in the same place in their user info, so there isn’t any change in practice to our simple endpoint.

**Adding the Github Authentication Filter**

The main change on the server is to add an additional security filter to handle the "/login/github" requests coming from our new link. We already have a custom authentication filter for Facebook created in our ssoFilter() method, so all we need to do is replace that with a composite that can handle more than one authentication path:

SocialApplication.java

private Filter ssoFilter() {

CompositeFilter filter = new CompositeFilter();

List<Filter> filters = new ArrayList<>();

OAuth2ClientAuthenticationProcessingFilter facebookFilter = new OAuth2ClientAuthenticationProcessingFilter("/login/facebook");

OAuth2RestTemplate facebookTemplate = new OAuth2RestTemplate(facebook(), oauth2ClientContext);

facebookFilter.setRestTemplate(facebookTemplate);

UserInfoTokenServices tokenServices = new UserInfoTokenServices(facebookResource().getUserInfoUri(), facebook().getClientId());

tokenServices.setRestTemplate(facebookTemplate);

facebookFilter.setTokenServices(tokenServices);

filters.add(facebookFilter);

OAuth2ClientAuthenticationProcessingFilter githubFilter = new OAuth2ClientAuthenticationProcessingFilter("/login/github");

OAuth2RestTemplate githubTemplate = new OAuth2RestTemplate(github(), oauth2ClientContext);

githubFilter.setRestTemplate(githubTemplate);

tokenServices = new UserInfoTokenServices(githubResource().getUserInfoUri(), github().getClientId());

tokenServices.setRestTemplate(githubTemplate);

githubFilter.setTokenServices(tokenServices);

filters.add(githubFilter);

filter.setFilters(filters);

return filter;

}

where the code from our old ssoFilter() has been duplicated, once for Facebook and once for Github, and the two filters merged into a composite.

Note that the facebook() and facebookResource() methods have been supplemented with similar methods github() and githubResource():

SocialApplication.java

@Bean

@ConfigurationProperties("github.client")

public AuthorizationCodeResourceDetails github() {

return new AuthorizationCodeResourceDetails();

}

@Bean

@ConfigurationProperties("github.resource")

public ResourceServerProperties githubResource() {

return new ResourceServerProperties();

}

and the corresponding configuration:

application.yml

github:

client:

clientId: bd1c0a783ccdd1c9b9e4

clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1

accessTokenUri: https://github.com/login/oauth/access\_token

userAuthorizationUri: https://github.com/login/oauth/authorize

clientAuthenticationScheme: form

resource:

userInfoUri: https://api.github.com/user

The client details here are registered with [Github](https://github.com/settings/developers), also with the address localhost:8080 (same as for Facebook).

The app is now ready to run and gives the user the choice between authenticating with Facebook or Github.

**How to Add a Local User Database**

Many applications need to hold data about their users locally, even if authentication is delegated to an external provider. We don’t show the code here, but it is easy to do in two steps.

1. Choose a backend for your database, and set up some repositories (e.g. using Spring Data) for a custom User object that suits your needs and can be populated, fully or partially, from the external authentication.
2. Provision a User object for each unique user that logs in by inspecting the repository in your /user endpoint. If there is already a user with the identity of the current Principal, it can be updated, otherwise created.

Hint: add a field in the User object to link to a unique identifier in the external provider (not the user’s name, but something that is unique to the account in the external provider).

**Hosting an Authorization Server**

In this section we modify the [github](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_github) app we built by making the app into a fully-fledged OAuth2 Authorization Server, still using Facebook and Github for authentication, but able to create its own access tokens. These tokens could then be used to secure back end resources, or to do SSO with other applications that we happen to need to secure the same way.

**Tidying up the Authentication Configuration**

Before we start with the Authorization Server features, we are going to just tidy up the configuration code for the two external providers. There is some code that is duplicated in the ssoFilter() method, so we pull that out into a shared method:

SocialApplication.java

private Filter ssoFilter() {

CompositeFilter filter = new CompositeFilter();

List<Filter> filters = new ArrayList<>();

filters.add(ssoFilter(facebook(), "/login/facebook"));

filters.add(ssoFilter(github(), "/login/github"));

filter.setFilters(filters);

return filter;

}

The new convenience method has all the duplicated code from the old method:

SocialApplication.java

private Filter ssoFilter(ClientResources client, String path) {

OAuth2ClientAuthenticationProcessingFilter filter = new OAuth2ClientAuthenticationProcessingFilter(path);

OAuth2RestTemplate template = new OAuth2RestTemplate(client.getClient(), oauth2ClientContext);

filter.setRestTemplate(template);

UserInfoTokenServices tokenServices = new UserInfoTokenServices(

client.getResource().getUserInfoUri(), client.getClient().getClientId());

tokenServices.setRestTemplate(template);

filter.setTokenServices(tokenServices);

return filter;

}

and it uses a new wrapper object ClientResources that consolidates the OAuth2ProtectedResourceDetails and the ResourceServerProperties that were declared as separate @Beans in the last version of the app:

SocialApplication.java

class ClientResources {

@NestedConfigurationProperty

private AuthorizationCodeResourceDetails client = new AuthorizationCodeResourceDetails();

@NestedConfigurationProperty

private ResourceServerProperties resource = new ResourceServerProperties();

public AuthorizationCodeResourceDetails getClient() {

return client;

}

public ResourceServerProperties getResource() {

return resource;

}

}

|  |  |
| --- | --- |
|  | the wrapper uses @NestedConfigurationProperty to instructs the annotation processor to crawl that type for meta-data as well since it does not represents a single value but a complete nested type. |

With this wrapper in place we can use the same YAML configuration as before, but a single method for each provider:

SocialApplication.java

@Bean

@ConfigurationProperties("github")

public ClientResources github() {

return new ClientResources();

}

@Bean

@ConfigurationProperties("facebook")

public ClientResources facebook() {

return new ClientResources();

}

**Enabling the Authorization Server**

If we want to turn our application into an OAuth2 Authorization Server, there isn’t a lot of fuss and ceremony, at least to get started with some basic features (one client and the ability to create access tokens). An Authorization Server is nothing more than a bunch of endpoints, and they are implemented in Spring OAuth2 as Spring MVC handlers. We already have a secure application, so it’s really just a matter of adding the @EnableAuthorizationServer annotation:

SocialApplication.java

@SpringBootApplication

@RestController

@EnableOAuth2Client

@EnableAuthorizationServer

public class SocialApplication extends WebSecurityConfigurerAdapter {

...

}

with that new annotation in place Spring Boot will install all the necessary endpoints and set up the security for them, provided we supply a few details of an OAuth2 client we want to support:

application.yml

security:

oauth2:

client:

client-id: acme

client-secret: acmesecret

scope: read,write

auto-approve-scopes: '.\*'

This client is the equivalent of the facebook.client\* and github.client\* that we need for the external authentication. With the external providers we had to register and get a client ID and a secret to use in our app. In this case we are providing our own equivalent of the same feature, so we need (at least one) client for it to work.

|  |  |
| --- | --- |
|  | We have set the auto-approve-scopes to a regex matching all scopes. This is not necessarily where we would leave this app in a real system, but it gets us something working quickly without having toreplace the whitelabel approval page that Spring OAuth2 would otherwise pop up for our users when they wanted an access token. To add an explicit approval step to the token grant we would need to provide a UI replacing the whitelabel version (at /oauth/confirm\_access). |

To finish the Authorization Server we just need to provide security configuration for its UI. In fact there isn’t much of a user interface in this simple app, but we still need to protect the /oauth/authorize endpoint, and make sure that the home page with the "Login" buttons is visible. That’s why we have this method:

@Override

protected void configure(HttpSecurity http) throws Exception {

http.antMatcher("/\*\*") **(1)**

.authorizeRequests()

.antMatchers("/", "/login\*\*", "/webjars/\*\*").permitAll() **(2)**

.anyRequest().authenticated() **(3)**

.and().exceptionHandling()

.authenticationEntryPoint(new LoginUrlAuthenticationEntryPoint("/")) **(4)**

...

}

|  |  |
| --- | --- |
| **1** | All requests are protected by default |
| **2** | The home page and login endpoints are explicitly excluded |
| **3** | All other endpoints require an authenticated user |
| **4** | Unauthenticated users are re-directed to the home page |

**How to Get an Access Token**

Access tokens are now available from our new Authorization Server. The simplest way to get a token up to now is to grab one as the "acme" client. You can see this if you run the app and curl it:

$ curl acme:acmesecret@localhost:8080/oauth/token -d grant\_type=client\_credentials

{"access\_token":"370592fd-b9f8-452d-816a-4fd5c6b4b8a6","token\_type":"bearer","expires\_in":43199,"scope":"read write"}

Client credentials tokens are useful in some circumstances (like testing that the token endpoint works), but to take advantage of all the features of our server we want to be able to create tokens for users. To get a token on behalf of a user of our app we need to be able to authenticate the user. If you were watching the logs carefully when the app started up you would have seen a random password being logged for the default Spring Boot user (per the [Spring Boot User Guide](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#boot-features-security)). You can use this password to get a token on behalf of the user with id "user":

$ curl acme:acmesecret@localhost:8080/oauth/token -d grant\_type=password -d username=user -d password=...

{"access\_token":"aa49e025-c4fe-4892-86af-15af2e6b72a2","token\_type":"bearer","refresh\_token":"97a9f978-7aad-4af7-9329-78ff2ce9962d","expires\_in":43199,"scope":"read write"}

where "…​" should be replaced with the actual password. This is called a "password" grant, where you exchange a username and password for an access token.

Password grant is also mainly useful for testing, but can be appropriate for a native or mobile application, when you have a local user database to store and validate the credentials. For most apps, or any app with "social" login, like ours, you need the "authorization code" grant, and that means you need a browser (or a client that behaves like a browser) to handle redirects and cookies, and render the user interfaces from the external providers.

**Creating a Client Application**

A client application for our Authorization Server that is itself a web application is easy to create with Spring Boot. Here’s an example:

ClientApplication.java

@EnableAutoConfiguration

@Configuration

@EnableOAuth2Sso

@RestController

public class ClientApplication {

@RequestMapping("/")

public String home(Principal user) {

return "Hello " + user.getName();

}

public static void main(String[] args) {

new SpringApplicationBuilder(ClientApplication.class)

.properties("spring.config.name=client").run(args);

}

}

|  |  |
| --- | --- |
|  | The ClientApplication class MUST NOT be created in the same package (or a sub-package) of the SocialApplication class. Otherwise, Spring will load some ClientApplication auto-configurations while starting the SocialApplication server, resulting in startup errors. |

The ingredients of the client are a home page (just prints the user’s name), and an explicit name for a configuration file (via spring.config.name=client). When we run this app it will look for a configuration file which we provide as follows:

client.yml

server:

port: 9999

context-path: /client

security:

oauth2:

client:

client-id: acme

client-secret: acmesecret

access-token-uri: http://localhost:8080/oauth/token

user-authorization-uri: http://localhost:8080/oauth/authorize

resource:

user-info-uri: http://localhost:8080/me

The configuration looks a lot like the values we used in the main app, but with the "acme" client instead of the Facebook or Github ones. The app will run on port 9999 to avoid conflicts with the main app. And it refers to a user info endpoint "/me" which we haven’t implemented yet.

Note that the server.context-path is set explicitly, so if you run the app to test it remember the home page is <http://localhost:9999/client>. Clicking on that link should take you to the auth server and once you you have authenticated with the social provider of your choice you will be redirected back to the client app

|  |  |
| --- | --- |
|  | The context path has to be explicit if you are running both the client and the auth server on localhost, otherwise the cookie paths clash and the two apps cannot agree on a session identifier. |

**Protecting the User Info Endpoint**

To use our new Authorization Server for single sign on, just like we have been using Facebook and Github, it needs to have a /user endpoint that is protected by the access tokens it creates. So far we have a /user endpoint, and it is secured with cookies created when the user authenticates. To secure it in addition with the access tokens granted locally we can just re-use the existing endpoint and make an alias to it on a new path:

SocialApplication.java

@RequestMapping({ "/user", "/me" })

public Map<String, String> user(Principal principal) {

Map<String, String> map = new LinkedHashMap<>();

map.put("name", principal.getName());

return map;

}

|  |  |
| --- | --- |
|  | We have converted the Principal into a Map so as to hide the parts that we don’t want to expose to the browser, and also to unfify the behaviour of the endpoint between the two external authentication providers. In principle we could add more detail here, like a provider-specific unique identifier for instance, or an e-mail address if it’s available. |

The "/me" path can now be protected with the access token by declaring that our app is a Resource Server (as well as an Authorization Server). We create a new configuration class (as n inner class in the main app, but it could also be split out into a separate standalone class):

SocialApplication.java

@Configuration

@EnableResourceServer

protected static class ResourceServerConfiguration

extends ResourceServerConfigurerAdapter {

@Override

public void configure(HttpSecurity http) throws Exception {

http

.antMatcher("/me")

.authorizeRequests().anyRequest().authenticated();

}

}

In addition we need to specify an @Order for the main application security:

SocialApplication.java

@SpringBootApplication

...

@Order(SecurityProperties.ACCESS\_OVERRIDE\_ORDER)

public class SocialApplication extends WebSecurityConfigurerAdapter {

...

}

The @EnableResourceServer annotation creates a security filter with @Order(SecurityProperties.ACCESS\_OVERRIDE\_ORDER-1) by default, so by moving the main application security to @Order(SecurityProperties.ACCESS\_OVERRIDE\_ORDER) we ensure that the rule for "/me" takes precedence.

**Testing the OAuth2 Client**

To test the new features you can just run both apps and visit <http://localhost:9999/client> in your browser. The client app will redirect to the local Authorization Server, which then gives the user the usual choice of authentication with Facebook or Github. Once that is complete control returns to the test client, the local access token is granted and authentication is complete (you should see a "Hello" message in your browser). If you are already authenticated with Github or Facebook you may not even notice the remote authentication.

**Adding an Error Page for Unauthenticated Users**

In this section we modify the [logout](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_logout) app we built earlier, switching to Github authentication, and also giving some feedback to users that cannot authenticate. At the same time we take the opportunity to extend the authentication logic to include a rule that only allows users if they belong to a specific Github organization. The "organization" is a Github domain-specific concept, but similar rules could be devised for other providers, e.g. with Google you might want to only authenticate users from a specific domain.

**Switching to Github**

The [logout](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_logout) sample use Facebook as an OAuth2 provider. We can easily switch to Github by changing the local configuration:

application.yml

security:

oauth2:

client:

clientId: bd1c0a783ccdd1c9b9e4

clientSecret: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1

accessTokenUri: https://github.com/login/oauth/access\_token

userAuthorizationUri: https://github.com/login/oauth/authorize

clientAuthenticationScheme: form

resource:

userInfoUri: https://api.github.com/user

**Detecting an Authentication Failure in the Client**

On the client we need to be able to provide some feedback for a user that could not authenticate. To facilitate this we add a div with an informative message:

index.html

<div class="container text-danger error" style="display:none">

There was an error (bad credentials).

</div>

This text will only be shown when the "error" element is shown, so we need some code to do that:

index.html

$.ajax({

url : "/user",

success : function(data) {

$(".unauthenticated").hide();

$("#user").html(data.userAuthentication.details.name);

$(".authenticated").show();

},

error : function(data) {

$("#user").html('');

$(".unauthenticated").show();

$(".authenticated").hide();

if (location.href.indexOf("error=true")>=0) {

$(".error").show();

}

}

});

The authentication function checks the browser location when it loads and if it finds a URL with "error=true" in it, the flag is set.

**Adding an Error Page**

To support the flag setting in the client we need to be able to capture an authentication error and redirect to the home page with that flag set in query parameters. Hence we need an endpoint, in a regular @Controller like this:

SocialApplication.java

@RequestMapping("/unauthenticated")

public String unauthenticated() {

return "redirect:/?error=true";

}

In the sample app we put this in the main application class, which is now a @Controller (not a @RestController) so it can handle the redirect. The last thing we need is a mapping from an unauthenticated response (HTTP 401, a.k.a. UNAUTHORIZED) to the "/unauthenticated" endpoint we just added:

ServletCustomizer.java

@Configuration

public class ServletCustomizer {

@Bean

public EmbeddedServletContainerCustomizer customizer() {

return container -> {

container.addErrorPages(new ErrorPage(HttpStatus.UNAUTHORIZED, "/unauthenticated"));

};

}

}

(In the sample, this is added as a nested class inside the main application, just for conciseness.)

**Generating a 401 in the Server**

A 401 response will already be coming from Spring Security if the user cannot or does not want to login with Github, so the app is already working if you fail to authenticate (e.g. by rejecting the token grant).

To spice things up a bit we will extend the authentication rule to reject users that are not in the right organization. It is easy to use the Github API to find out more about the user, so we just need to plug that into the right part of the authentication process. Fortunately, for such a simple use case, Spring Boot has provided an easy extension point: if we declare a @Bean of type AuthoritiesExtractor it will be used to construct the authorities (typically "roles") of an authenticated user. We can use that hook to assert the the user is in the correct orignization, and throw an exception if not:

SocialApplication.java

@Bean

public AuthoritiesExtractor authoritiesExtractor(OAuth2RestOperations template) {

return map -> {

String url = (String) map.get("organizations\_url");

@SuppressWarnings("unchecked")

List<Map<String, Object>> orgs = template.getForObject(url, List.class);

if (orgs.stream()

.anyMatch(org -> "spring-projects".equals(org.get("login")))) {

return AuthorityUtils.commaSeparatedStringToAuthorityList("ROLE\_USER");

}

throw new BadCredentialsException("Not in Spring Projects origanization");

};

}

Note that we have autowired a OAuth2RestOperations into this method, so we can use that to access the Github API on behalf of the authenticated user. We do that, and loop over the organizations, looking for one that matches "spring-projects" (this is the organization that is used to store Spring open source projects). You can substitute your own value there if you want to be able to authenticate successfully and you are not in the Spring Engineering team. If there is no match, we throw BadCredentialsException and this is picked up by Spring Security and turned in to a 401 response.

The OAuth2RestOperations has to be created as a bean as well (as of Spring Boot 1.4), but that’s trivial because its ingredients are all autowirable by virtue of having used @EnableOAuth2Sso:

@Bean

public OAuth2RestTemplate oauth2RestTemplate(OAuth2ProtectedResourceDetails resource, OAuth2ClientContext context) {

return new OAuth2RestTemplate(resource, context);

}

|  |  |
| --- | --- |
|  | Obviously the code above can be generalized to other authentication rules, some applicable to Github and some to other OAuth2 providers. All you need is the OAuth2RestOperationsand some knowledge of the provider’s API. |

**Conclusion**

We have seen how to use Spring Boot and Spring Security to build apps in a number of styles with very little effort. The main theme running through all of the samples is "social" login using an external OAuth2 provider. The final sample could even be used to provide such a service "internally" because it has the same basic features that the external providers have. All of the sample apps can be easily extended and re-configured for more specific use cases, usually with nothing more than a configuration file change. Remember if you use versions of the samples in your own servers to register with Facebook or Github (or similar) and get client credentials for your own host addresses. And remember not to put those credentials in source control!

Want to write a new guide or contribute to an existing one? Check out our [contribution guidelines](https://github.com/spring-guides/getting-started-guides/wiki).

|  |  |
| --- | --- |
|  | All guides are released with an ASLv2 license for the code, and an [Attribution, NoDerivatives creative commons license](https://creativecommons.org/licenses/by-nd/3.0/) for the writing. |

**Get the Code**

HTTPS SSH Subversion



[DOWNLOAD ZIP](https://github.com/spring-guides/tut-spring-boot-oauth2/archive/master.zip)

[GO TO REPO](https://github.com/spring-guides/tut-spring-boot-oauth2)

**Table of contents**

* [Single Sign On With Facebook](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_simple)
* [Add a Welcome Page](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_click)
* [Add a Logout Button](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_logout)
* [Manual Configuration of OAuth2 Client](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_manual)
* [Login with Github](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_github)
* [Hosting an Authorization Server](https://spring.io/guides/tutorials/spring-boot-oauth2/#_social_login_authserver)
* [Adding an Error Page for Unauthenticated Users](https://spring.io/guides/tutorials/spring-boot-oauth2/#_custom_error)
* [Conclusion](https://spring.io/guides/tutorials/spring-boot-oauth2/#_conclusion)
* [TEAM](https://spring.io/team)
* [SERVICES](https://spring.io/services)
* [TOOLS](https://spring.io/tools)
* [STORE](https://store.spring.io/)
* [NEWSLETTER](https://spring.io/blog)

© 2018 [Pivotal Software](https://www.pivotal.io/), Inc. All Rights Reserved. [Terms of Use](https://pivotal.io/terms-of-use) • [Privacy](https://pivotal.io/privacy-policy) • [Trademark Guidelines](https://spring.io/trademarks)