|  |
| --- |
| Course Overview  Course Overview  Hi, everyone. My name is Wojciech Lesniak, and welcome to my course, Authentication and Authorization in Spring Boot 3 Using Oauth2. I am a technical lead with over 10 years experience securing applications with Spring Security. Spring Security can take away the complexity of securing your applications with Oauth2. In this course, we are going to leverage Spring Boot 3 to quickly and effectively configure Spring Security Oauth2 in your application. Some of the major topics that we will cover include authentication in the client with OpenID Connect, configuring an Authorization Server, securing a resource server with roles, scopes and claims, implementing common security patterns in OAuth2. By of this course, you'll know how to configure Spring Security OAuth2 using Spring Boot 3. Before beginning this course, you should be familiar with the Spring Framework. I hope you'll join me on this journey to learn how to configure OAuth2 using the Spring Framework with the Authentication and Authorization in Spring Boot 3 Using Oauth2 course at Pluralsight.  Configuring the Authorization Server  Introduction  Hi, it's Wojciech Lesniak. Welcome to my course, Authentication and Authorization in Spring Boot 3 Using Oauth2. This course is focused on implementation, so I won't be going into too much detail into the theory of OAuth2. If you're not familiar with OAuth2, don't worry, you'll still be able to follow along. And the Pluralsight library has courses that take a more deeper dive into OAuth if you need more clarification on something. Let's first do a version check. This course was created by using Spring Boot 3, Spring Security 6, and Java 17. If you're using Spring Boot 2, then this course is less applicable. However, I have a course in the Pluralsight library that you can check out for that. Next, let's get started on the first module.  The Authorization Server  OAuth2 comprises of a number of actors, the subject, client, Authorization Server, and the Resource Server. To illustrate how they interact, in our demo application crypto portfolio, our subject will be the user Victoria.  Victoria uses the crypto portfolio web application to keep track of her crypto asset performance. Her personal details and portfolio is called the protected resource in OAuth2. The Resource Server is what hosts and protects the protected resource.  To access her portfolio, Victoria uses a client, which could be a webpage via a browser or an app on her phone. The client needs access to Victoria's data from the Resource Server. The Resource Server will not give access to the protected resource unless the client has a valid access token.  This is where the Authorization Server steps in.  The client will act as the relaying party and will redirect to Victoria to the Authorization Server for authentication.  The **Authorization Server** needs to do three things, authenticate Victoria, authenticate their client, and get confirmation from Victoria that she actually gives permission to this client to access her portfolio data.  Then, it needs to issue an access token to the client.  Once the client has the access token, it can then use it to make a request to the Resource Server to get Victoria's portfolio data and only Victoria's portfolio data.  No other person's.  Now, there are **many options for an Authorization Server.**  There are Identity Providers as a Service, like Okta and Auth0, where you just pay a subscription, and they manage everything for you.  Now, most cloud platforms will also have an Authorization Server, like Azure AD or AWS Cognito.  You could also install and deploy and manage your own out‑of‑the‑box solution like **Keycloak**. And Red Hat also offers a paid solution, which provides support for your organization if you need that, too. And then, finally, there is also more customizable solutions where you have to build it yourself, like the **Spring Authorization Server,** which is quite new to the Spring ecosystem and has only recently made GA.  For this, you will need to do a lot of configuration and coding to get one production ready. Hence, in the real world, you will most likely use one of the other options , I mentioned. However, since this is a Spring course, I feel that I have to cover the Spring Authorization Server. However, we will also use Keycloak.    ====================  [Setting up Spring Authorization Server]  Like all things Spring Boot, getting started with an Auth Server is super easy. I'm going to use the Spring Initializr via the IntelliJ plugin. However, you can also use the one on the web. Let's give the project a name. Let's change the package. We then selected the following dependencies, Spring Authorization Server, Resource Server, Spring Security, and web. That's it. And as you can see, the following dependencies have been added to Maven, and we can now start a brand new Authorization Server. Easy. Now let's configure some of our actors. Let's start with Victoria.  In the application properties, I actually like to use YAML instead. Just a personal preference. Let's register her under spring, security, user. We can give her a username and password. Okay, Victoria is now registered and has a way to authenticate with the Auth Server.  Now, this is definitely not production ready. You wouldn't just have the plaintext password here. It would, at the minimum, be encrypted. And also, we'd probably store it in a secure datastore.  But, in the early stages of development, this is a great way to get up and running. The next actor is the client. In our properties under spring, security, oauth2, authorizationserver, here, we can configure all things Authorization Server. So let's start with the client, give it the name and the client‑id and secret. Again, the curly brackets, noop, basically means that we're using no password encoder, effectively plaintext. The next property is the client‑authentication‑methods. We're going to set these to client\_secret\_basic. This property configures how the Auth Server authenticates the client. In this case, it will expect a basic authorization header containing the client‑id and secret. Now, Spring also supports form authentication. With the support of grant‑types, we will use the authorization code, as this is the most common and secure flow. You will see it in action once we test this. We can also set some scopes. For now, we're going to set openid connect to make this an OpenID Connect flow and one custom scope, portfolio. Now OpenID Connect is designed for authenticating users in browser‑based applications, which is actually perfect for the use case we have now. Open ID Connect actually builds on top of OAuth2, as OAuth2 by itself is naturally an authorization protocol, not an authentication protocol. The next property, require‑authorization‑consent. If you've ever signed in with Google via third party using OAuth, you've probably seen these pop‑ups asking if you give permission for the client to access your data. Now, this is what this is. After Victoria authenticates with the Auth Server, it will ask her for consent to the scopes requested by the client. So let's just set this to true for demo purposes. Okay, one last step. Since we're going to test this with Postman, we need to add the Postman callback URL via the redirect‑uris field. Now, this essentially is a whitelist of all the callback URLs the Auth Server will accept for this client. It's a way to protect against spoofing, you know, someone injecting another redirect URL. Let's switch over to Postman, which will play the role of the clients, to test the Authorization Server config. We're going to create a new request. Now we don't need any val URLs. We just want to get a token, as we haven't actually created the Resource Server yet. Under Authorization, OAuth2, our client needs the following fields. By the way, as you can see, this is where we got the Callback URL from. So let's set the client‑id and secret from application.yaml file. We'll set the Scope to OpenID Connect. For Client Authentication, set to Send as Basic header. This is what we configured in our Auth Server. For the remaining fields, because we're using OpenID Connect, we can get the data from the discovery document. Now, the discovery endpoint will always have the path of .well‑known‑openid/configuration. This document essentially allows the client to automatically bootstrap itself. So let's add the Auth and access token URLs into Postman. Okay, let's try and get an access token. You can see, we now have a browser pop up with the login page from the Authorization Server. If we look at the browser history, you can see that our client Postman redirected us here. It set the response\_type to code, which is the authorization code flow, provided its client\_id to identify itself, the state, scopes, and the redirect URL. Basically, everything we configured. It's like the client is saying, hey, Authorization Server, I'm the portfolio client. I have a user claiming to be Victoria. She wants to access her portfolio. I'm going to forward her to you so you can authenticate her. Once you're done, redirect her back to me to this URL with the authorization code. Cheers. So, once Victoria authenticates, you can see the Auth Server redirected her browser back to the client with the authorization code and the state parameter. And here is the access token. Now to get the access token to the client, Postman, in this case, had to actually make a back channel request directly to the Spring Authorization Server to exchange the Auth code for the access token. The client also had to include its client secret to authenticate itself. This may seem overcomplicated, but there's a purpose here. The Auth code was sent via the browser i.e., the front channel, which is not very secure. But by itself, it's useless without the client's secret. So that's okay. The access token here is actually a JSON Map. It's Base64 encoded. If you go to jwt.io, you can actually decode this and see what's inside. And as you can see, in the body of the message, the following, subject, Victoria, we have some of the scopes that we requested, expiry date, and some other fields. Okay, next, let's configure the same for Keycloak to see just how easy it is to do it with an out‑of‑the‑box solution.  Authenticating with KeyCloak  Keycloak is available as a standalone version you can download, a Docker image, Kubernetes, or OpenShift. All the details are on the Keycloak web page. I'm going to download the stand‑load version. All I need to do is unzip it. Then, from the bin folder, I can start Keycloak with the following command, which basically optimizes it for local development, and I'm going to make a start on the following port. If I access the admin console on localhost and the port we selected, I get prompted with admin login. So, I'm going to log in. And here, is where we can configure everything. Now, the first thing we need to do is create a realm. A realm in Keycloak is basically a level of isolation. You can create many different types of realms, and all the realms are isolated from each other. There's no interaction between the realms. They have their own users, clients, and settings. So, each realm is kind of like a separate identity provider or Authorization Server. You know, think of realms as kind of like tenants in a multi‑tenant application. There is one exception. There is one special realm called the master realm. Now, this is reserved for managing the other realms. Also, your super admins are created here. Now, these admins have permission to view and manage all the other realms, so best practice is to never create any clients or regular users in the master. So let's create a new realm for crypto‑portfolio. Done. We can now switch over to our realm and create the user Victoria. Under users, we can fill in the details. As you can see, we have a rich set of options, a lot more than a Spring Authorization Server, and a nice GUI to do it, things like such as email verification, etc. So let's give her some credentials. So in the Credentials tab, we can set her password here. Okay, now let's create the portfolio client. Our realm has a Clients section. Here, we can create the same client we did in the Spring Authorization Server. We give it a name, Next, Client authentication, On. This configures OpenID Connect and makes this a confidential client. In Auth, there are two different types of clients, confidential and public clients. The main difference is public clients don't have the capability to store safely a client‑id and secret. So, think of all of clients that are in the browser, right? Whereas, confidential clients are more sort of backend server side, so they can have a client‑id and secret and keep it secure. Only Standard flow for now, which is basically just the authorization code flow. Under redirect URIs, we will enter the Postman callback URI. Again, this is the whitelist for all the redirect URIs Keycloak will accept. Okay, that's it. Our client is created nice and easy, and all via the console. And as you can see, we're only scratching the surface. Keycloak is very configurable or via the GUI. And in addition, all our credentials are stored securely and encrypted by Keycloak. You would need a lot of coding to get the same functionality in the Spring Authorization Server. So let's test this in Postman. Now, since we have a confidential client, we need a client's secret. Under the Credentials tab, we can generate one. As you can see, we're only scratching the surface of all the features that are in Keycloak. Let's do a quick test in Postman. If we go to our Realm settings, we can see the OpenID Connect endpoint. And as you can see, just like with the Spring Authorization Server, it ends with .well‑know/openid/configuration. Let's create another request and fill in the Keycloak details. And just like for the Spring Authorization Server, we are redirected back to Keycloak where Victoria can log in. And Voila, we have an access token. Nice and easy. Now, with our Auth Service set up, let's wrap up this module.  Wrap-up  In this module, we covered the Authorization Server, its purpose, your options, and how to set one up, giving you a single point of control. It also isolates user data and credentials, which limits access to that data, so injection vulnerabilities introduced in an application feature doesn't lead to a data breach of user data. Now, Spring Authorization Server is still relatively new. And as you saw, you have to do a lot of configuration and coding to get it anywhere near production ready, like securely storing user and client credential data, creating a user registration page, or client registration page. And most of these things you can already get out of the box in more mature products like Keycloak. So, why reinvent the wheel? But it's still a great tool for local development or very simple service‑to‑service authorization. Next, let's secure the client.    ========================  (3.) Creating a Client Application  Introduction  Hi, it's Wojciech Lesniak. In this module, we're going to focus on the clients. Crypto portfolio requires a new front end so that the users can view their portfolios. This web application will be the client in the OAuth2 flow. It will redirect the user to the Authorization Server for authentication and consent, then exchange the Auth code for an ID and access token. Now there are two different types of clients, confidential and public. Confidential clients are able to securely hold a client's secret and keep it safe. Now typically, these are your multi‑page applications, which run server side, while your public clients cannot hold a client's secret securely. These are typically your single‑page applications running on the user's browser or on their desktop. In this module, we're going to learn the best practices of dealing with public and confidential clients and how to configure a new OAuth OpenID Connect client using Spring Boot.  Authenticating Confidential Clients  In this demo, we're going to use Spring Boot to quickly set up a new web application and secure it with OAuth2. Now I'm going to use IntelliJ again and use Spring Initializr. I'll give it a name. Now for the Dependencies, I'm going to select Spring Web, as this is going to be a Spring Web MVC application. And then for Security, I'm going to select OAuth2 Client. For now, we'll just create a simple HTML page with a Welcome greeting. Now, typically, when developing a new application, in the past security was often a bit of an afterthought. It was typically rushed and hacked into the project at the later stages. Now this often resulted in a suboptimal security solution. But now, with Spring Boot, we can secure this web app right from the get go. In the overflow, this web app would represent the client, as it would delegate authentication of users to the Authorization Server. In the previous modules, we registered a client for crypto‑portfolio in both Keycloak and Spring Auth Server. Now I'm going to use Keycloak. But in the demo code, there will be the configuration for Spring Auth Server. To configure the OAuth2 client, in the application properties file, I'm going to use a yaml file under client, registration. And since this will be a confidential client, we need to add the client‑id and secret. In Keycloak, we can get that from the Credentials tab of the client. If we want to use Spring Auth Server, then the client‑id and secret is what we configured in the application.yaml file. So let's add the client‑id and secret. Let's also add the redirect‑url, the grant‑types, which will be the Auth code flow. The scope will have OpenID Connect, as that's how we want to authenticate. Basically, most of the details that we entered into Postman in the previous module. Now, what we need to do is provide the provider details. The provider is basically the Authorization Server. So in the configuration under provider, all we need to do is add the issuer‑uri. Now this is because we're using Open ID Connect, so our client can access the discovery documents on the Authorization Server. And here, all the information required to bootstrap itself is located, so things like the token and authorization endpoints. We also need to link this provider with our clients. That's it. If we restart our application and try to access it, you can see we are redirected back to the Authorization Server. If we log in as Victoria, the Authorization Server then redirects us back to the application. Now, this is far from production ready. But, then again, neither is our web app. However, our security implementation can now evolve along with the application development, which, in the end, will most likely result in a more robust, cleaner security solution. Next, let's look at how we can customize this security implementation to make cryptos portfolios unique security requirements.  Customizing the Client  Let's fast forward to the development of our application. Voila, Victoria can now view and monitor her crypto portfolio positions. So far, Spring Boot has set up our entire security implementation, and it did this via the order configuration classes. If we take a peek under the covers into our Spring Boot library under autoconfigure, security, clients, and servlet, here are all the order configuration classes for our OAuth2 clients. Ours is a web app, so let's open the OAuth2 web security configuration. This class is activated on the condition that there is a client registration, which we configured in our application.yaml file. It then creates the various components we need, like the AuthorizationClientService, the repository, and, most importantly, the SecurityFilterChain. Now Spring Boot's configuration requires that all requests are to be authenticated and that authentication will be done via OAuth2. In the background, Spring Boot configures a filter, the DelegatingFilterProxy. Now this filter will intercept all requests in the web server before they reach any servlets. It then delegates the requests to the appropriate SecurityFilterChain. The filter chains, as the name suggests, are a collection of filters that perform various security checks and tasks and can even allow the request to proceed to the next filter in the chain or block it. If we put a breakpoint in the FilterChainProxy and make a request to our web application. In debug mode, we can see the security filter chain. Our request is in the process of passing through. Here we have a filter that handles logging out. If our request is not authenticated, then the Auth Redirect Filter will forward us to the Authorization Server, and OAuth filters will handle things like the callback from the Authorization Server. If you recall, the Spring Boot configuration required that all requests are to be authenticated, so the Authorization filter at the end will enforce this. Now, as you can see, if we want to customize our authentication solution in any way, then we need to be able to manipulate this filter chain and the filters within. Now to do that, we can create our own SecurityConfiguration class. We annotate it with the @Configuration and @EnableWebSecurity. We then create a method that returns our new SecurityFilterChain. The method will accept an builder object as a parameter. And just like Spring Boot did, we can use this builder to customize our filter chain. If we go back to the Spring Security implementation, you can see it's annotated with ConditionalOnDefaultWebSecurity. And if we drill into this annotation, it has a condition that the SecurityFilterChain bean must be missing before it activates. Now, this makes sense. If we don't configure our own SecurityFilterChain, then Spring Boot would create one for us. And since now we are exposing a SecurityFilterChain as a bean, Spring Boot will back off. Now, at the moment, our configuration is very similar to Spring Boot. All requests need to be authenticated, and we're going to use oauth2Login. However, one change we are going to make is to add an authenticationSuccessHandler to the oauth2Login. Now, this is a class that implements the authenticationSuccessHandler interface and overrides the unauthenticationSuccess method. And this will be called every time a user successfully logs in via OAuth, as Crypto Portfolio needs to be able to register these users and create a portfolio for them. And using the SuccessHandler, we can actually do that for the user without them having to fill in any registration forms. Now, as you can see, this method is exposed to the Authentication object, and we can use that to get the user claims, like their username, email, name, etc. If you recall, previously, I mentioned OpenID connect is an open‑standard for decentralized authentication, which basically means it's an identity layer built on top of OAuth. So when the user successfully authenticates with OpenID Connect, the client can exchange the Auth code for an access token and an ID token. Now the access token is used for authorization to access any protected resources, so its intended audience is the Resource Server. The ID token, on the other hand, is for the client. It represents the authentication event that happened and contains information about it and about the user's identity. The client can use the ID token for things like displaying the user's name on the page. Or, in the case of Crypto Portfolio, to automatically register the user. So it's very important to remember that the ID token is for the client and should not be sent to any APIs. And if you look at our authentication object, you can actually see that we have access to this ID token. And here you can see all the claims about the authentication event. The Issuer, which is basically the Authorization Server that issued this ID token. The Subject, in OAuth, the subject is the user or entity that is being authenticated. The Audience, who this token is issued for. In this case, it will be the Crypto Client app. When the token expires and when it was issued. Additionally, you can also have claims about the user, like their name, email, etc. And you can configure what is on the ID token at the Authorization Server. Now, best practice is to only include the bare minimum required by the client. Now, this will reduce the impact of any ID tokens being leaked, as these tokens are by value and are required to be in JSON format so they can be read by anybody. And then if you need any more information about the user, you can call the UserInfo endpoint. Spring actually does this automatically for us, and we can access the user info on the Authentication object. Now, we can access the Authentication object in other parts of the code, and we can do this by using the static THREADLOCAL class, the SecurityContextHolder. We can get the context and then get the authentication. Additionally, in our controllers, we can access the principal by adding it as a method parameter and anotating it with the AuthenticationPrincipal. Next, let's take a look at public clients.  Securing Public Clients  In the previous clips, we covered confidential clients. But a lot of modern web applications are now single‑page browser‑based application written in some sort of JavaScript framework, like React or Angular. Now these come with a number of security challenges. The main one is there is no way to securely store the client‑id and secret, as the code is running in the user's browser, and anyone can see it. In the Auth code flow, if you recall, the client secret was used to authenticate the client when it was exchanging the Auth code for the access token. Now, one way to solve this in a single‑page application is to use the backend‑for‑frontend pattern, where the backend handles the Open ID Connect authentication flow for the frontend, and all tokens, like the ID token and the access token are stored server side. The frontend is simply secured by cookies, or you can use the Auth code flow with Proof of Key Exchange. Now, this is a very brief intro into how to secure public clients, as it can be a whole course in itself. In fact, I actually have a course that covers this in more detail, Using Spring Security OAuth2 Login. Now, this covers how you can secure a single‑page application using the backend‑for‑frontend pattern and also covers Proof of Key Exchange in detail. So if you're interested, check it out. But next, let's wrap up this module.  Wrap-up  In this module, we cover the client in the OAuth2 flow. It is important to remember there are different types of clients, public and confidential. Each has its pros and cons and each needs to be secured differently. For confidential clients, we use the Auth code flow. For public clients, if we're handling the OAuth2 flow in the frontend, the browser, then we need to use the Auth code flow with Proof of Key Exchange. The implicit flow should not be used. In fact, if you're not using open ID Connect, then it is recommended that you also use Proof of Key Exchange, even in confidential clients. Next, let's take a look at the Resource Server.  Creating a Resource Server  Introduction  Hi, it's Wojciech Lesniak again. In this module, we're going to cover the Resource Server. If you recall, the Resource Server is the entity that hosts the protected resource. Currently, Crypto Portfolio is a monolith. The architectural team have decided it needs to be split up into microservices to improve the scalability amongst other things. Hence, we'll have a frontend as the client, the Authorization Server, and now a separate microservice for the backend, which will be our Resource Server, as this is where the user's portfolio data will be stored. Now, we need to secure the communication between the client and the resource server using OAuth2, so let's configure the Resource Server.  Creating a Resource Server  Our new microservice, the portfolio service is a Spring REST API. In order to secure it with OAuth2, we need to add the Spring Boot's data dependencies, Spring Boot's data security and Spring Boot's oauth2‑resource‑server. Next, we need to configure our filter chain to use OAuth2 authorization with JWT tokens. We can do this, again, by creating a Configuration class that exposes a bean of type SecurityFilterChain. Our filter chain will require all endpoints to be authenticated and that we will be using OAuth2 with JWT authorization. So, using the oauth2ResourceServer method on the HTTP security object, we can access the ResourceServer configuration via a lambda function. Here, we can set that we want JSON Web Tokens. The JWT configuration can also be customized using a lambda function. But for now, we will use the Customizer.withDefaults. This indicates that we want the default settings. The access token is issued by the Authorization server. Now, JWT tokens are basically a Base64‑encoded string and comprise of three sections separated by a period, the header, the body, and the signature. The signature can be used to verify that token has not been tampered with and, most importantly, that it was issued by the Authorization Server. Now, in order for our Resource Server to verify the signature, it needs the public key of the Authorization Server. Now, this information is on the discovery document of the Authorization Server. So, all we need to do is in the application.yaml under spring, security, oauth2, resourceserver, jwt, is at the issuer‑uri. And again, that's in the discovery document as well. Now, our Resource Server has everything it needs to bootstrap itself and load the public keys. That's it. If we start our Resource Server, you can see any calls to the API now result in a 401 Unauthorized. Next, let's write some test cases for our security.  Testing Your Security  It is also very important to test your security, just like you would do your business logic. Ideally, you would use a test‑driven development approach. Spring Boot and Sprint Security make this very simple. All we need to do is add the following dependencies to our project, spring‑boot‑starter‑test and spring‑boot‑security‑test. Next, we need to configure a test profile and a test configuration file, where we can configure the typical things we would need for testing, like an in‑memory database and our test security configuration. Now when it comes to validating tokens, we don't want to start up an authorization server every time we run our unit tests. That just would be overkill. So what we can do is create a private and public key pair. I'm going to add them under the test resource folder, as this is only used during testing. Then, under security, oauth2, resourceserver, jwt, rather than adding an issuer‑uri, we're going to set the public‑key‑location, which is basically the location of our pem file in the class part. Now our test Resource Server can now use this key to verify the access token's signature. And in our PortfolioControllerTest class, we can create a standard WebMveTest. We set the ActiveProfiles to test so that it overrides the configuration to use the public key we provided. Now, for our first test case, we're just going to call the Portfolio endpoint using the mockMvc object, just like any controller unit test you would have probably would have written. And here, we should expect an isUnauthorized. Now, our next test will include the valid access token. So we need to create a Test token, which will be in JWT format. Now, if you drill in, you can see that we created a KeyPair from the public and private pem files. We created a header and a body. Now, in the body of our JWT token, we can add test user's details, so basically the subject and the username. And then, finally, we just sign it with the private key. We can now include this access token in the header. The name will be AUTHORIZATION, and the value will be Bearer, space, access token. Now, since we have signed it with the private key, the Authorization Server should be able to verify the signature using the public key we provided in the application.yaml file. And if we run our tests, you can see they are passing now. As you can see, there is not much to it. You can include authentication and authorization tests alongside your integration unit tests. A big culprit of security violations is actually misconfigurations. Hence, this is a great way to test your security configuration. Okay, now that we have a token, let's look at how we can access some of the claims in the token next.  Accessing the Principals JWT Claims  Crypto Portfolio is now made up of two applications, the frontend and the backend microservice. Now, when Victoria accesses the frontend, the frontend will include the access token in all requests to the Resource Server. Hence, in our Resource Server, if we want to access some of these claims, like get the user's username, we can use the AuthenticationPrincipal annotation on our controller methods. And since we have configured for JWT access tokens, this object will be of type Jwt. And we can now extract the subject claim from the token, which is effectively the username, and it's required to be unique. Hence, no need to pass in a username via the header, query stream, or request body. And this makes it much more secure, as a client can only retrieve the portfolio for users it has a valid token for. And the Authorization Server typically requires the user consent before they can get this token. So let's put a breakpoint here and see this token. You can see we have access to the header, the body, the claims, and the actual token string itself. Now we can access the Principal object anywhere in the code, not just in the controllers, and we can do this by the SecurityContextHolder object. The @AuthenticationPrincipal annotation is just a much more cleaner and convenient way to do the same in our controllers. Again, you can configure as much or as little claims to be added to your JWT token via the Authorization Server.  Switching to an Opaque Token  Up until now, Crypto Portfolio was using JWT tokens, which allow us to verify the token offline by checking the signature with the Authorization Server's public key. This approach is more performant and efficient. However, it does have some challenges. The size of the token can actually impact performance, and there is actually a limit to the size. Additionally, there is no way to invalidate the token after it has been minted. The Resource Server will continue to accept it until it expires, even if it's invalidated at the Authorization Server. In some very high‑sensitive applications, we also might not want any claims to be written on the token, and we might want to verify it with the Authorization Server each time. Now, in OAuth2 terms, this is known as an opaque token. So let's configure our application so that for each request, it verifies the token with the Authorization Server. In our security configuration, we're going to change opaqueToken, rather than a JWT. For now, we will be using the default settings. If we go to our application.yaml file under resourceserver, the resourceserver needs to know three things. It needs to know the client ID and secret and the introspection endpoint URI. If we go to Postman, we can actually make a request to this introspection endpoint. We can get the endpoints details from the discovery document. You just need a valid token, client ID, and secret. And Voila, the token is verified. You can see now why this approach is more secure. If the token was leaked or fell into nefarious hands, it's useless without the client ID and secret. And for a successful request, we also get a response with all the user's claims, and we can now access this token via the SecurityContextHolder. Now, this will give us access to all the claims that were returned by the introspection endpoint via the getTokenAttributes. Crypto Portfolio just needs the user's username, so we're going use the subject field. Again, it really depends on your use case. And most of the time, verifying tokens via the introspection endpoint is simply overkill, and the JWT token with a short expiry time is usually more preferred. However, in some sensitive operations, the additional hop of verifying the token for each request is worth the performance here. Like, for example, transferring cryptos, just gives it that added security.  Making the Resource Server Stateless  By default, Spring configured the portfolio service to be stateful. If we make a request to our servers via browser, you can see it actually returned a cookie with a JSESSIONID. This is then forwarded to the server along with the bearer token. Crypto Portfolio architecture requires the microservices to be scaled horizontally. Maintaining state between microservices, especially if you need to scale horizontally, is challenging, so it's best to be avoided. The JWT token effectively plays the role of a distributed cookie, as all the claims we need are on the token, and it is forwarded to the microservice for authorization in each request. Hence, in our security configuration, we can configure the session management and set it to use STATELESS. Now there's no overhead in dealing with sessions in our application. However, it's worth noting that there is a limit to the number of claims that can be added to a token, as it's effectively transported via the header. If we make the request again, we can see now that the server does not return the JSESSIONID cookie. Next, let's wrap up.  Wrap-up  For the most part, using their JWT token setup allows for offline stateless authentication and authorization. It is also a great way to propagate user context in stateless services. For most use cases, this is fine, especially if the expiration time is kept to a minimum. However, there are use cases where for sensitive operations, you might require an opaque token, or you might want to perform token introspection via the Authorization Server. Spring Security caters for all such scenarios. And finally, it's important to also test your security to reduce the risk of configuration changes resulting in security breaches. In the next module, we'll cover authorization in more detail using scopes and roles.    =========================    Implementing Scope and Role-based Authorization  Introduction  Hi, it's Wojciech Lesniak again. In this module, we are going to use scopes, roles, and authorities to secure Crypto Portfolio. It can be a little bit confusing as to the difference between each and when one should use them. Scopes are part of the OAuth2 specification. OAuth2 is essentially a delegated authorization protocol where a client acts on behalf of a resource owner to access the protected resource. The resource owner can restrict what the client can access on their behalf by consenting to scopes. For example, a scope of email, name means that the client can only retrieve these details about the user i.e., their email and their name. Roles, on the other hand, are assigned to a user to determine what privileges they have. So, for example, if a client has an access token for Victoria with the scope of portfolio‑view, even though Victoria has an admin role assigned to her, the client can only view her portfolio, as that is the scope that was consented to and is on the access token. Vice versa, if a client is assigned the scope of portfolio‑admin, but Victoria does not have the admin role, then the client cannot perform admin tasks on her behalf. That's essentially what delegated authorization means. Next, let's configure some scopes.  Authorization with Scopes  Crypto Portfolio now has a new microservice, the support service. This service provides an API for users to raise support issues and for administrators to view and respond to. Now the business wants to add a new mobile client. Users can use this client to view their portfolios. However, administration functionality is to be disabled for mobile and only available in web. Hence, in our support service SecurityConfiguration, using the requestMatchers, we can configure that for all requests to support /admin API endpoints. The clients require the SCOPE\_portfolio.admin. We can do this by the hasAuthority method. And as you can see, we need to prefix it with SCOPE\_. Now the reason for this prefix is that Spring keeps authorities, roles, and scopes under the same collection. And in order to differentiate between them and to prevent name collision, scopes are prefixed with SCOPE\_ and Roles with ROLES\_, while authorities have no prefix. So now, if we go back to Keycloak, in our application realm, we can create a new client scope, portfolio.admin, and then assign it to the crypto‑portfolio client, which is used by our web application. Now we need a new client for mobile. All the settings will be the same, but we won't add the portfolio.admin scope for this client. If we try to access the admin API via Postman, you can see the client is not authorized, as it does not have the admin scope, regardless if the user is an administrator or not. If we take a look at the token minted for the web application, you can see the new scope is under the scope claim. Spring Security automatically maps all the scopes in this claim into granted authorities with the scope prefix. Next, let's look at configuring roles.  User Authorization with Roles  In the previous clip, we restricted access to the client via scopes. However, if a user is not an administrator, they should not have access to the administration section of the web application, additionally, to the API, as admins can view all support queries. You could use scopes to restrict access. There is actually nothing stopping you from doing this in Spring. As mentioned before, Spring just stores this in the granted authorities. But ideally, we should use a role, as this is for assigned user privileges. First, let's create a new admin role. In the Keycloak console under Realm roles, we will name it portfolio‑admin. Next, let's create a user henry. Henry will be our administrator, so we'll give him a password, and then assign the admin role to him. Okay, now if I'm into token for henry, you can see this role is in the claims under realm\_access roles. Great. Now let's go back to the SecurityConfiguration class for the support service. Our requirement for Crypto Portfolio is that the client has to have the scope portfolio admin, and the user must also be an admin, hence, have the admin role. Under our admin request matcher, the options we have is either one authority or has any authorities. Any accepts multiple authorities, but only one condition needs to be met, and we need both of them to match. If we add another request matchup for the same path, the first one that evaluates to true will authorize access. So essentially, it's like the has any authority, as Spring evaluates them in order. Hence, we need to use the access method, which takes in an AuthorizationManager. We need two authorization managers, one for the scope and one for the role, so we can wrap them in an AuthorizationManager allof function. This says that all of the auth managers need to evaluate a true. One last thing is missing. If we look at the token, the portfolio\_admin scope is under scopes, so Spring can automatically map this to the granted authorities. But the user roles are nested under realm\_access. Now this is not part of any OAuth2 specification, so we need to specifically map these to the role. By default, Spring uses a class, the JwtAuthenticationConverter class, which converts the JWT token into an Authentication object. The Authentication object has the collection of authorities. And if you recall, it's placed in the SecurityContextHolder. Hence, this class uses another converter, the JwtGrantedAuthoritiesConverter, which, as the name suggests, converts JWT claims into granted authorities. Now, by default, it simply maps all the values in the scope field into the granted authorities collection as scopes. Hence, we can create a converter class that converts from JWT into a collection of granted authorities. In the Converter method, we override, we can get the realmAccess claim, get the list of the roles, and return a list of granted authority. Now because these are roles, we're going to prefix them with role\_. Now we can wire this converter into our SecurityConfiguration, and then create a method that returns a new instance of the JwtAuthenticationConverter, and simply pass in a new custom GrantedAuthoritiesConverter. Then, using the JWT configured, add our new custom.JwtAuthenticationConverter to the filter chain. Well, that's it. If we start our application and try to access the admin API with an access token from Victoria, we get an unauthorized as she is not an admin. However, if we try with henry, then we are authorized, as he has the admin role. Next, let's look at more finer‑grained authorization.  Method-level Authorization  Spring also supports more finer‑grained authorization at the method level. Let's look at the admin controller. This method returns all support queries for all users. The web client uses it to populate the admin screen with all the user support queries, allowing for admins to respond. In our security config, we configured to access to this URI requires both the admin scope and role. We can actually do the same check at the method level using the Preauthorize annotation. This annotation accepts a Spring expression, so we can use the hasRole, operator, and hasAuthority. Now these checks may seem redundant, as we're already doing that in the SecurityConfiguration. But it can add that additional layer of protection and in case the developer accidentally misconfigurations the security config. Now the Preauthorize can perform checks before a method was called. But we can also check the return object before a method returns. In our SupportQuery controller, this method returns a query response. This time using the PostAuthorize, we can check that the return object's username matches the one in the token. Now, this ensures that we are only returning the user sensitive data, not any other user, and this can be just an additional layer of protection in case of a bug or SQL vulnerability accidentally exposes other users' data. Now, Spring expressions are actually quite powerful. Here, we have a method that returns queries based on the userId parameter. This is the subject in the access token. Now this method is used by the admin and non‑admin pages. The security requirements is that admins can return queries for any user, but non‑admins can only return their own support queries. This makes this method very vulnerable to injection or request tampering. So, we can use a preauthorized expression that either the access token has the scope and role of admin, or the access token's subject matches the requests userId parameter. Next, let's look at another layer.  Verifying the Token Audience  Currently, we are just checking the signature of a token to see if it's valid. That might not be enough, especially if the Authorization Server means tokens for different applications. An access token should only be used for its intended audience. There is a field on a token named just that, audience. You can see the token Keycloak minted for Victoria has the audience and its portfolio‑service. But now that we have two services, let's add the support service. We first create a scope and then set up a mapper for the audience. Then, in our application.yaml file for the support service under audiences, we can add portfolio‑service. And the same for the support service, we can add the support‑service audience. Now, if mobile client attempts to call the support service, even if the client had the scope for portfolio‑admin and the user the role of admin, the request will be denied, as the audience is for the portfolio‑service only. Next, let's wrap up this module.  Wrap-up  In this module, we looked at configuring authorization with Spring Security 3 and OAuth2. A key takeaway is that it's important to have a defense in‑depth security strategy. In OAuth, that's verifying the token signature, audience, the request, and method invocation. This way, any misconfigurations at one layer does not result in a security breach, as the other layers still provide protection. In the next module, let's look at security in the call chain.    ===================  Securing the Call Chain with OAuth2  Introduction  Hi, it's Wojciech Lesniak again. The Crypto portfolio application is growing. There is now a new Pricing microservice and multiple clients. This brings with it additional security challenges, such as additional endpoints to secure, service‑to‑service authorization, and securing the microservices from both internal and external parties. Now, fortunately, there are security patterns we can leverage to address this using the Spring Security framework.  API Gateway with Spring Cloud Gateway  Typically, you would not expose your services externally. They would sit behind a firewall protected by the network, and then all access to the service would be via an API Gateway. Now, this has a number of benefits. All internal service endpoints are not exposed externally, only the ones that need to be, and the API Gateway can also perform authorization and authentication checks. So let's configure an API Gateway. Now, the Spring framework has one that we can use, and it's part of the Spring Cloud framework. Using the Spring Initializr again, we can select the Spring Cloud Gateway dependency and the Spring Boot Starter OAuth2 Resource Server Dependency. Now, by adding the Resource Server Dependency, the Gateway can perform authorization checks on the access token before routing them to any other service. If we go to the YAML file of our new API Gateway, first, we need to create a route to the target service. Let's create one for the portfolio service. We put the URI of the service and a predicate. Basically, anything to path portfolio will be routed to the web clients, and anything to this support path will go to the support service. Now, just like any other resource server, we also need to add the issuer‑uri so that this service can bootstrap itself with all the OAuth2 configuration. Okay, let's give this a test. If we make a request to the API Gateway, you can see that we now get the portfolio details back. The Gateway performed authorization checks and then routed the request to the portfolio service. Now, typically in most microservice systems, this would be the approach. All the services sitting behind a firewall with a Gateway exposing the API. Next, let's look at Identity Propagation.  Identity propagation with Token relay  When Crypto Portfolio was a monolithic application, the user context could be easily retrieved from the session. Now, the user context is actually on the access token or can be retrieved from the user‑info endpoint using the access token. Hence, we can pass the access token along the call chain. This is known as token relay. In fact, we are already doing this in the API Gateway, as it automatically forwards the access token to the portfolio or support services. Now the new pricing service is the furthest down the call chain. We also need to secure this service. So just like any Resource Server, we have the resource‑server security dependencies for Spring, we provide the issue the URI, and we also require the pricing‑service audience. Now in Keycloak, under the existing portfolio‑service scope, we can create a new mapper for the support service audience. This will be a custom one, and we also want it to be included in the token. Okay, if we mint the token for Victoria, you can see that the token has an audience of portfolio, support, and pricing‑service. Hence, it can be passed around to all of these services, providing authorization and user context propagation. Now, in the portfolio‑service, we use the web client to make a call to the pricing service. We need to include the access token along with the request, otherwise, the request will be unauthorized. We could do it manually, getting the token from the SecurityContextHolder and setting it in the header, but there is an easier way. When we build the web client, we can add a filter, the ServletBearerExchangeFilterFunction, and Spring will handle propagation of the token for us. Next, let's have a look how we can actually make the pricing‑service even more secure.  Client Credentials with Web Client  If we take a look at the pricing‑service, all it actually does is return the price of crypto. It doesn't actually need most of these claims on this token. And if this service was ever hacked or the token leaked, then it could be used to gain access to the other services. Additionally, we are also making a call to the price service every portfolio request. A future plan feature for Crypto Portfolio is for the portfolio service to schedule periodic calls to the price service and cache the prices, hence, reducing the number of calls to the price service and improving performance. The portfolio service needs a way to mint its own access tokens, and we can use the OAuth2 client credentials grant to achieve this. It is designed for service‑to‑service authorization. In Keycloak, we need to create a client for the pricing service. This client will only have client authentication. Now the client credential grant is very simple. There is no browser or user consent involved. The client simply sends a request to get a token, providing its client‑id and secret. So Crypto Portfolio services is now also a client, hence, we need to add the Spring oauth2‑client dependency. Then, in the properties, we need to register the new clients. We'll give it a name, set the grant type to client\_credentials, and add the client‑id and secret. Portfolio service now needs to make a request to the token endpoint of the Authorization Server to get the new access token. Fortunately, Spring has a filter for the web client that can handle this for us. Okay, let's see if I can get the name of this in one take, ServletOAuth2AuthorizedClientExchangeFilterFunction. Now this will automatically call the token endpoint to retrieve a token for the pricing‑service and include it in the request. It will actually cache the token for future requests and get a new one if the token is close to expiration. We create an instance of the ServletOAuth2AuthorizedClientExchangeFilterFunction. The function requires a client repository, so we can just wire that in, and pass it into the constructor along with the client repository. Since we have one client, we can actually set the client‑id to be default. That's it. Spring will handle everything for us. Nice and easy. Now, if we have multiple clients, then when making the web client call, we would need to set an attribute to let the ServletOAuth2AuthorizedClientExchangeFilterFunction know which client it's making the request for. That's it. Nice and easy. Now, let's look at token exchange to make our architecture even more secure.  Token Exchange  If we review Crypto Portfolio's architecture, you can see we have a nice defense in‑depth security implementation. The web client uses Open ID connects to authenticate users. All the microservices are behind an API Gateway. We currently have token relay, so each service is performing authorization and authentication checks. If we look at this token, the audience effectively is for three services, the web, portfolio‑service, and support service. Now, since we are using token relay, all the roles, scopes, user claims, and audiences need to be on this token to satisfy all services. If the token was ever leaked due to one of the services being compromised, then practically everything is compromised. There is another new approach, Token Exchange. Basically, the portfolio web client could exchange its token with the Authorization Server for a portfolio‑service token or a support‑service token and each time the new token's audience would be limited to the audience of the target service. Now currently, support for Token Exchange is in its infancy. Spring doesn't yet support it out of the box, neither does the Spring Authorization Server, but Keycloak does have support for it in preview mode. However, that's not going to stop us from implementing it. We'll create some new clients, as now all our services need to have their own client. So we'll create a portfolio web, portfolio service, and a support service client. Let's start with the portfolio‑web‑client. We're going to select Client authentication, Authorization, and we're just going to have the Standard flow. We will add the Postman redirect for testing. And because this is a relaying party, we'll add its own redirect URL as well. We can now change the client‑id and secret for our portfolio‑web‑service in the Spring application.yaml file. Next, for the portfolio‑service, we need to remove the support‑service audience we added earlier, as well as the pricing one. So the token should only have the portfolio‑service audience. That's it. If we now mint this token, you can see it has the user's claims and only its own audience. Now, we have to give permission for the portfolio‑web‑service to exchange its token for the portfolio‑service‑token. In the portfolio‑service config under Authorization, Policies, we create a policy for the Client, which will simply return true if the token being exchanged was minted for the portfolio‑web‑service. Now let's go to Permissions. You should see token‑exchange. If you don't see it, you need to restart Keycloak with the preview‑feature parameter. We select our policy. This is basically saying, if the client is the portfolio‑web‑client, then it can exchange an access token for the portfolio‑service‑token. Okay, let's test this. If you look at the RFC for Token‑exchange, it's quite simple, a call to the token endpoint with a form body with the following fields. The request is basically saying, hey, Authorization Server, I want to exchange this token for the portfolio‑service token, and here is my client‑id and secret to prove my identity. If we run it, you can see we get back a new access token for the portfolio‑service. We can now do the same configuration for the support‑service. So now, in the API Gateway, we can create the data‑transfer objects for the Token Exchange request and response. Now we just need a Gateway filter. A Gateway filter in the API Gateway is a filter that you can use to edit the response before it leaves the Gateway after it's been routed. Now inside this filter, we're basically doing the same thing as the Postman request. We make a POST request to the Authorization Server and exchange tokens, providing all the fields that are required. The filter then switches the token in the header before it sends out the request to the portfolio‑service or the support‑service. If we look at our architecture, we have unique tokens for each service. Again, for most cases, this is probably overkill, and adds additional hops to the Auth server for each request, so there is definitely a performance here. But for some sensitive applications, this adds that additional layer of security. Next, let's wrap up the module and the course.  Wrap-up  This concludes the module and the course. As you can see, there is no one way to implement OAuth2 with Spring Security. Now, typically, JWTs with short expiration times will cover most of your use cases. However, as your application grows, it could lead to token bloat. It's important to remember that the claims you put on the token about the user could be sensitive. And if it's leaked, it could result in the data breach of sensitive user data, so at least, you should encrypt it. But ideally, you should keep the token claims to a bare minimum. Now, a defense in‑depth strategy is key. Services should sit behind an API Gateway and only the necessary API should be exposed externally. Each service should perform its own authorization and authentication checks, which include checking the tokens signature, the audience, the scopes, and the roles. And Spring also supports even finer‑grained authorization checks at the request and method invocation layer. Opaque tokens and token exchange give that added layer of security, but do come with additional complexity and a performance hit. Thanks for taking the time to watch my course, and feel free to post any comments or any questions you have. |