Course Overview

Hi everyone. My name is Wojciech Lesniak. Welcome to my course, Effective OAuth2 with Spring Security and Spring Boot. OAuth2, OpenID Connect, and JSON Web Tokens make it easy for your users and services to securely access your application without having to remember or provide their passwords. That's powerful and can significantly improve user experience, streamline your registration process, and make your application more secure. In this course, we are going to dive into Spring Boot and Spring Security 5's support for OAuth2 and OpenID Connect. Some of the major topics we will cover include what JSON Web Tokens, OAuth2, and OpenID Connect is and is not and what security challenges they can help you solve, how to add social sign-in to complement your application's existing authentication, how to secure both public and confidential clients, securing distributed systems and the various microservices patterns you can leverage. By the end of this course, you'll have the foundational knowledge of OAuth2, OpenID Connect, and JWT and the practical knowledge of how to implement it in your application in Spring Boot and Spring Security. Before beginning the course, you should be familiar with Java and the Spring framework. I hope you'll join me on this journey to learn Spring Security OAuth2 with the Effective OAuth2 with Spring Security and Spring Boot course at Pluralsight.

Spring Security Oauth2: The New Direction

Introduction

Hi. It's Wojciech Lesniak. Welcome to my course on Effective OAuth2 with Spring Security and Spring Boot. Security can be difficult and challenging as it's constantly evolving and difficult to keep up with. Hackers are always trying to come up with new and innovative ways to get around your application's security. So why reinvent the wheel? With Spring Security, you're joining a large community of developers constantly innovating and updating the framework to your and their applications. In this course, we'll go over Spring Security support for OAuth2, OpenID Connect, and JWT, which is an exciting and growing space within cybersecurity. And don't worry if you're not familiar with these standards. By the end of the course, you will be. Now this is not going to be your typical happy flow course where we do a few simple Spring starter projects with OAuth2 enabled. My goal is to first provide you with the foundational knowledge of OAuth2, OpenID Connect, and JWT. Then I'll cover various scenarios you might face in real life, like using OAuth2 and OpenID Connect to add social sign-in to an existing application or how to use solely OAuth2 or OpenID Connect for single sign-on in all your applications. We will cover Spring Security support for server-side and public clients, like single-page applications, and how to effectively secure distributed applications, which might be using microservices and the various security patterns you can implement. We'll end with advanced customization and scope-based authorization. So let's get started.

The challenges with Authentication / Authorization in modern applications

If you've been using Spring Security before, you'll probably be familiar with the following approach: a login page prompting the user for their credentials, by default the username and password. This is then transmitted to the server in the body of the request or in the header, like basic authentication over HTTPS hopefully. Server-side, the password hash is retrieved from the database and matched against the hash of the password in the authentication request. If successful, a session is created with the authenticated user, and the cookie with the session ID is returned to the client. All subsequent requests to the server include the session ID to avoid having to re-authenticate. Now this is very easy to set up with Spring. However, this approach has the following challenges. The user's credentials are handled by the application and transmitted in plain text, hence an increase risk of being leaked by some malicious code. Also, storing credentials alongside application data exposes them to any injection vulnerabilities introduced by the development team. It also doesn't scale very well as the number of applications and services in your organization grows as each application will need its own often duplicated authentication and identity management. Often using different technology stacks requiring maintenance on all the login registration pages and keeping up to date with security best practices and updates becomes difficult. Users also need to maintain multiple sets of credentials. And since they tend to reuse the same passwords, your organization security becomes as strong as the weakest link. In essence, some sort of single sign-on is required where a user authenticates once with an identity provider and then gets the keys to access all the applications they are authorized for and not just within your organization's boundaries. If a potential user already has an account with a third party like Google, how can Google and your application trust each other to allow an authenticated user with Google to authenticate and share the identity in Google with your application? This is very common now, sign in with Google or Facebook, etc. The user can then agree to share their identity details, which can then be used by your application to register them automatically without having to fill out any time-consuming registration forms or simply authenticate and allow them to access your application without handling or storing their passwords. OAuth2, OpenID Connect with JWT are standards that can help solve this. Additionally, most applications start out as monoliths with a UI, service, and data access layer. These are straightforward to implement and secure. You only need to protect a UI or API entry point. However, they don't scale well. There is a limit at how much hardware you can throw at them to meet increased demand. Eventually, you will need to break them up into microservices and scale horizontally. Now from a security standpoint, you will now have more endpoints to secure. A request from a user might require multiple hops to complete. How can you authenticate a request between hops without sharing the password and without a session? How can you maintain the identity state of a user between stateless autonomous services? And how can you do it without impacting the performance of the application? In a monolith, latency is low due to interprocess communication. In a distributed system, there is now the additional cost of the network hop. Hence, how can you minimize any additional latency of any security checks? Next, we will look at ways how tokens can help address some of these issues.

Why tokens

To explain tokens, let's start by describing the alternative. Here we have a web application and a number of microservices. You could secure the communication between the browser and the web server or edge service, which will be exposed externally and leave the internal services behind a firewall unsecure. To maintain state, you could include user info, like the username in the body or query string of the requests. However, you're leaving yourself exposed to anyone who breaks into your network or anyone working in your organization. There have been a number of high-profile hacks that were perpetrated by insiders, like contractors or consultants. You could pass the user credentials along with each request. A popular approach is via basic authentication. This is basically a Base64 encoder string of the username:password, which is added to the header of the request with the authorization property with Basic space encoded string. These would be passed down with each request. And each endpoint would authenticate the username and password with perhaps an identity provider like LDAP. However, this approach doesn't scale. Each hop requires a call to the identity provider to authenticate the credentials and retrieve the identity details. And too many services are handling credentials, increasing the chance of them being leaked. To improve this, after authentication with credentials is performed with the identity provider, an identifier, like a GUID, is returned, which could then be sent instead. The identifier is a pointer to the state in the identity server. It can then be passed around to other services. The services can verify and source the identity details from the provider. This solves the password problem as it's no longer handled by the services. This is essentially a token. You would sent it in a similar manner as the basic authentication, just with the Bearer prefix. However, in its current form, it's no different than a password. If a token gets leaked, it could be used to access anything just like the password. In fact, it might be worse than a password because the user cannot simply change it, and it doesn't solve the scaling problem due to the identity checks at each hop. This is why tokens also have an expiry time associated with them to limit damage if exposed. Now this type of token is referred to as a by reference token. It is a reference to the resource. Another type of token is a by value token, which will include the state information within in, like the username, email, and what the bearer of the token is authorized for, hence eliminating the need to call the identity provider. These tokens are generally signed by the provider. Hence, services can use the public keys of the provider to ensure it is valid and hasn't been tampered with. For most use cases, a by reference is fine, and the additional hops to the identity server has the benefit of allowing you to invalidate the identifier with the identity provider. By value is generally preferred as it saves you the hop to the identity provider. However, anyone with access to the token has the visibility of the contents. And in certain circumstances, you might not want that as the data could be sensitive or give insight to the types of roles authorizations in your application. This can, however, be solved with encryption or the use of opaque tokens instead. The format of a by value token can be whatever you require. However, what should you use? The answer is a standard format as you don't want to be in a situation where every application you're integrating with requires a different way of passing, validating, and translating the token. So what standard formats are there? In the 90s, it was Kerberos. Early 2000s, SAML. And today, it's JSON Web Token, pronounced as jot. However, I always get it wrong and refer to it as JWT. Sometimes, by mistake, you might even catch me referring to it as Java Web Tokens. Now JWTs are in JSON format, SAML in XML. This makes JWTs more lightweight, compact, and easier to pass and hence mobile-friendly. It is digitally signed to prevent it from being tampered with and is Base64-encoded, so you can put it anywhere even in a URL unlike SAML. It's also protocol-agnostic. With SAML, you need to use SAML server and protocol. Whereas JWT doesn't mandate how you use it.

Introducing Json Web Tokens (JWT)

Previously, we had an opaque token, now a JSON Web Token. You're probably thinking the first one is better. It's smaller. Hold that thought. As you can see in its small format, it's simply a string as the JSON is Base64 URL-encoded. This is so you can transmit it easier in the header and even in the URL. Although if the payload is large, it can exceed the maximum length of a URL or header. The message is split into three parts separated by a period. If we decode it, we can see the two JSON maps and the signature. The first part is the header, which specifies the algorithm used to sign it with JSON Web Algorithms. JWA essentially is a registry of standard names and algorithms. So anyone reading this will know which algorithm the token is signed with and be able to easily verify it. Next is the payload. The JWT specification doesn't mandate what you need to put in here. It's solely based on your requirements. Finally, the JSON Web Signature, which uses the algorithm defined in the header. It's basically the signature of the header and payload. As mentioned, the JSON Web Token doesn't mandate what fields you can and cannot put in the payload. However, it does have standard optional fields, which you should definitely use. Now the best way to think of a JWT is a signed cookie. So with a JWT, a mobile user can authenticate with an identity provider. The identity provider will gather the user information and permissions into a JSON map, sign it, create a JWT, and send it to the client. So now rather than the state being in the server, it's client-side. The client then includes it in every request it makes. That's powerful. Firstly, if you have tens of thousands of users signed in on a mobile app making hundreds of calls per minute, having your server stateless and not maintaining and looking up state scales really well. As you can see, JWT in itself is a collection of standards from Java Web Algorithms, Java Web Signatures, and even Java Web Encryption where you can encrypt the content. Basically, it's a standard token format. Hence, we need other standards, like OAuth2, to define how to use the tokens for delegated access or which identity fields are required for federated access, such as OpenID Connect. In the next section, we'll describe how OAuth2 and OpenID Connect achieve this.

A journey into Oauth2

OAuth2 is an open standard used to protect a resource known as the protected resource. To understand OAuth, you need to understand the actors. Let's use a railway analogy to help explain this. A passenger wants to ride a train from Lewisham station to London Bridge station. In OAuth, the passenger would be known as the client, the entity that wants to access the protected resource. The train trip is the protected resource. Now the train trip is protected by the train station. In order for the passenger to get on the train, they need to get through the gates. In OAuth, the train station would be known as the resource server, an entity capable of authorizing access to a protected resource. In order to get through the gates, the passenger needs a valid ticket. In OAuth, that would be known as the token. It can be a by value where the details are visible on the ticket or by reference, like an eticket where the details of the trip are stored on the server and need to be looked up to validate. In order to get a valid ticket, the passenger needs to use a ticket machine. The ticket machine in OAuth is known as the authorization server, the entity that can issue the tokens. However, before it can issue the ticket, it needs to check with the train company on the price and if they authorize access to the trip. The train company is the resource owner, an entity capable of authorizing access to a protected resource. Now understanding these actors is key. And OAuth2 has a number of flows that orchestrate the interactions of these actors that cater to various scenarios. The most popular and secure is the authorization code grant. Let's go through it with an example.

Autherization Code Grant

Victoria wanted to encrypt her holdings with our demo app, Crypto Portfolio. Facebook and Crypto Portfolio have a trust relationship, i.e. Crypto Portfolio is a registered client with Facebook and has a unique client ID and client key. Hence, Victoria can authorize Crypto Portfolio to perform actions on her Facebook account on her behalf, such as access her profile data, post to her wall, or access her friends list. Crypto Portfolio will display the option you have probably seen before, like log in with Facebook. If Victoria clicks on it, Crypto Portfolio will send a redirect request to Facebook's authorization server via her browser. This begins the OAuth process with Facebook. First, Facebook will need to authenticate Victoria. If she is not already logged in with Facebook, she will need to do so. Once authenticated, Facebook's authorization server, using the client_id and scope, sees that the client, Crypto Portfolio, wants to access her profile and email. At this point, it will ask Victoria for her consent. You have probably seen these before clearly stating that Crypto Portfolio wants access to her profile and email. In this case, Facebook even gives the option for her to edit and remove certain scopes. If Victoria agrees, Facebook will then send a redirect response to the client via her browser with the URI specified by Crypto Portfolio and an authorization code. The authorization code can now be used by the client to exchange for the access token. The authorization code is a random number and effectively is a sender constraint token, meaning only the client can use it to get the access token as the resource server will also expect the client_id and secret be included with the request to retrieve the access token. Access tokens also have a very short expiry time. The reason for using an authorization code and not passing the access token directly is that transmission via the front channel between the authorization server, browser, and client cannot be completely relied upon as secure, especially the browser. It could have some malicious plugin. So sending the bearer access token would be risky. An authorization code, on the other hand, is useless without the client's secret, and that exchange happens via the back channel where the client goes directly to the authorization server. There is one other token that can be returned along with the access token, and that is the refresh token. Because access tokens are bearer tokens, they should have a very short expiry time. This is to limit the damage of an access token being leaked. However, often clients need the access token for longer and don't want to keep bothering the user to reconsent each time the token expires. This is where the refresh token comes in. It can be used by the client to get a new access token. Refresh tokens need to be stored securely on the client. If they are leaked, they can be used by a malicious party to keep re-requesting access tokens. Hence, ideally they should also have an expiry date and be single use, meaning each time a new access token is retrieved, a new refresh token is also issued. There are other grant types like client credentials for machine-to-machine authorization where the client is also a resource owner, is not a user, but an application or a service or implicit where you have a public client like a single-page application that cannot store the secret securely. We will also go over these in more detail in later modules. Now there is a lot of confusion around what OAuth actually is. The name adds to this ambiguity. Does it mean authentication or authorization? It's actually neither. We will discuss why in the next section.

Oauth2 is not Authentication

OAuth is actually for delegated authorization, not authorization. It is a resource owner giving a third party access to a protected resource on their behalf. The best way to get your head around this is with an example. Now if Victoria has a bank account, she is authorized to withdraw money from her account by the bank. That's authorization. However, if Victoria wants to delegate Adam to withdraw some funds from her bank account, she can authorize Adam to do so. However, that doesn't mean that Adam is authorized to withdraw the money by the bank on Victoria's behalf. If the bank has a policy that only the account holder is allowed to withdraw funds, even if the bank verifies with 100% certainty that Victoria authorized Adam to withdraw funds on her behalf, the transaction is still not authorized as it breaks the bank's policy. That's delegated authorization. Additionally, there is a misconception that OAuth is purely for social sign-in. That's one popular use case for OAuth. However, OAuth is much more. In fact, it's really not for authentication. We'll discuss why next.

Why we need OpenId Connect

Why do we need OpenID Connect? Well, because OAuth is not a federation, authentication, or identity protocol, yet it seems to be one of its most popular use cases. Firstly, let's define identity. Digital identity is a set of attributes related to an entity. An entity can be a human, organization, device, a service. An entity can have multiple digital identities, which are differentiated by the attributes associated with each. Take, for example, Victoria. Her identity at work as an employee is different to her identity at her fitness center, as a customer, or to her friends. Hence, depending on the application she is using, she will have different identities and restrict what attributes are available to them. Now in OAuth, the access token is intended for the resource server, not the client. However, because often the tokens are by value, they can be read by anyone and often contain identity information intended for the resource server, like a username to identify the resource owner. And developers on the client side often notice these details on the access token and use them to identify the user for authentication. The issue with this approach is that firstly, the resource owner might not have consented these details to be used by the clients. If the provider made a change to the access token format, it would break the client. I've seen this happen, and often the client's dev team would complain when, in fact, it's their fault. OAuth clearly states that the access token is intended for the resource server, not the client. They should simply pass the token on to the resource server to access the resource. Or the client will use the access token to retrieve the user's profile info and authenticate the bearer of the token as the user. However, OAuth is a delegation protocol. If Victoria gives her car keys to valet supervision to park her car who then delegates the task to another valet, having the keys in their possession doesn't make the valet Victoria. An access token can be passed around. So you can't assume that the bearer is the user. A malicious client could persuade a user to sign in to their application with a provider, then use the token against another client who also uses the same provider, effectively impersonating the user. Since the token was valid, the client would authenticate them. This exploit impacted a popular social media platform. Many providers solve these problems by building on top of OAuth. However, this means that you have to know each provider's implementation to integrate them with your application. To resolve this, OpenID Connect was created to consolidate all these lessons learned and best practices into a single standard to add authentication and identity to OAuth2.

What makes OIDC great for Authentication

One of the key new features of OpenID Connect is the identity token. It's a JWT that thinks claims about how the authentication occurred for the end user by the authorization server and allows your client to validate that authentication. Now to do this, it has a number of mandatory claims about the authentication process, such as the subject, which asserts the identity of the user. Now it must be unique and never reassignable. This allows a client to confidently use that field as a unique identifier for the user. The issuer field, this describes who's the issuing authority, and it allows the client to verify the user was authenticated by the expected source. Audience, this is a very important field. It basically identifies the client that was authenticated. Now it must be a client ID. The reason for this field is that your typical OpenID Connect provider, like Google, will support thousands of clients. Hence, your client can use this claim to ensure that the user was authenticated for your application. The expiration time, now one great feature about including this is that they also specify the format of the expiry time that the provider must use, which is great as before you had to deal with all the various different formats for each provider. Now there were also some useful optional claims as well. However, you generally find that most authentication providers will include these. Some key ones is the issued at, when was this token issued, and the auth time, when was it authorized. These are great for customizing your validation of a token. So for example, you might decide that even though the token is not yet expired according to the expiration claim, your application will only accept tokens that have been issued within the last 2 minutes. So it allows you to have your own expiration policy rather than relying on the authorization server's. Now OpenID Connect also introduced some standard scopes, and some key ones are the OpenID, which is mandatory and must be included as the first scope and indicates that this application intends to use OpenID Connect to verify the user's identity. Some of the other keys ones is a profile indicates that the user's profile claim is required, email address, and, of course, offline access, which is for the refresh token. Now you could also include custom scopes. Now there are many more other scopes. But the key feature is that for each scope, there are default standard claims associated with them. So if you user consents to a scope, the authorization server will return all or a subset of the standard claims. Now this is a great feature. You will see the benefit of it in the next module. Basically, if your application is looking for their user's email claim or the address and you're using OpenID Connect, then you know that that claim will be named address and email, not ADR or ADDR or EML or some other format the provider might use. Now additionally to the scopes, there are standard endpoints. So once you know the provider's location, the endpoint URLs will be the same for every OpenID Connect provider. Now some of the key endpoints are the UserInfo endpoint where the client can request previously consented user claims using the access token. You're probably thinking if we have an ID token, why do we still need the access token? Well the ID token might not have all the claims about the resource owner that were requested as it might be a thin token, meaning it has a subset of only the essential claims you might require for authentication. The ID token only mandates claims about how the authentication occurred and what format the user's claim should be in if the authorization server decides to add them to the ID token. Now this is generally to keep the size of the ID token small so it's compatible with mobile clients. However, if you need all the user's claims like say it's the first time the user is accessing your application and you need to register them, then you can go to the UserInfo endpoint and get all the claims. However, that's an implementation decision with the provider whether they provide a fat or thin token. The main point is the access token combined with the refresh token allows for rerequesting approved user claims without having to bother the user to reconsent each time. The WebFinder endpoint, this provides a way to dynamically discover the OpenID Connect provider for a given user. Provider metadata, this is a very useful endpoint. It returns a discovery document with all the OpenID Connect details of the provider and what it supports, such as support response, grant type, URL of the JW keyset, the asserted issuer, supported scopes, token endpoints, basically everything a client might need to dynamically configure itself with the provider and validate tokens. There's also a session management endpoint, which is quite useful. This checks if the user still has an active session with the authentication server as the client might also want to log the user out from their end if they log out from the provider. There are many others. I encourage you to check the spec. But the great thing is if you're dealing with an OpenID Connect provider, the endpoints follow a standard. So that makes them super easy to find. Next, let's look at the differences between OAuth2 and the OpenID Connect flow.

OIDC Authorization Code Grant Flow

OpenID Connect is built on top of OAuth2. So we have the same actors, the client, the resource server, the authorization server, and the resource owner. Let's go through a quick example. Now we're going to switch to Google as they are an OpenID Connect provider. Facebook uses their own implementation of OAuth2 called Facebook Connect. It's very similar to OpenID Connect. Fortunately, Spring Security actually support's Facebook's implementation right out of the box. Now if we quickly go through the flow, after Victoria selects Sign in with Google, the main difference is the redirect from the clients will now contain the OpenID scope followed by all the optional or custom scopes. The authorization server will now know to use the OpenID Connect flow. If the user consents, the client gets an authorization code, which he then uses to exchange for the access token and additionally now the ID token and optionally, of course, the refresh token. The ID token is for the client to use to verify the authentication while the access token can be used against the UserInfo endpoint to retrieve any additional information or claims about the resource owner. And the refresh token can be used to request a new access token once it expires. Now the above flow we described is the authorization code grant. In addition to this, there is the implicit flow and hybrid flow. We will go into more details around these flows in the upcoming modules when we discuss public clients. Next, we will talk about the radical change the Spring team have taken to support JWT, OAuth2, and OpenID Connect in Spring.

Spring Security 5 the new direction

Currently, OAuth2 support is scattered between a number of projects in Spring. If you look at the Spring OAuth2 feature matrix, there are four Spring projects with OAuth2 support, Spring Boot OAuth2, Spring Cloud Security, Spring 1, Spring Security OAuth 2, and Spring Security 5. You've probably also heard of Spring Social. This made things very confusing as to which one to use, and it depended on your requirements. Hence, the Spring team have decided to unify JWT Jose, OAuth2, OpenID Connect support under Spring Security proper as of Spring 5, which actually makes a lot of sense. Effectively, it's rewrite. Now currently, there still isn't full- feature parity. If you go through the matrix, you can see the gaps. But the plan is to unify all the projects under one. So if your feature is supported by Spring Security 5 and above, then use it as the other projects are in maintenance mode until the features are migrated to Spring Security. Okay, great. Now that's the theory out of the way. Let's get started into implementing OAuth2 and OpenID Connect in our demo application in the next section. If you have used Spring before to add a login page to your application and thought it was easy, you will be very pleased to know it's even easier to implement OAuth and OpenID Connect with Spring Security. The Spring team have done a great job.

Server-side Applications: Single Sign-in with Oauth2

Introduction

The demo application, Crypto Portfolio, allows users to monitor their cryptocurrency positions. In its current state, it uses a traditional three-tier architecture built using Spring MVC and Spring Boot, your typical monolith. It is secured with Spring Security and has its own user registration. If you have watched my previous course, you're probably familiar with this application. Data shows that a significant percentage of potential new users are not completing their user registration process, and there is an increased volume of failed login attempts and password resets. To address this, we can try to complement the site with social sign-in, giving users with either a Google or Facebook account the option of registering and logging into Crypto Portfolio without having to fill in the registration page or provide their password. Let's start with Facebook.

Registering the client with Facebook and Google

Facebook is not an OpenID Connect provider. They have their own implementation of Facebook Connect. So we will need to use the OAuth2 flow. Facebook will play the role of the authorization server and resource server, and Crypto Portfolio will be the client. If you recall back to the OAuth2 flow in the previous module, we need a client id and client secret. Hence, we need to first register our application with Facebook. We can do this by Facebook developers. Once registered, select create new app. Enter the display name and click Create App ID. Select Integrate with Facebook Login, then in PRODUCTS Facebook Login, Quickstart, select Web. Put in your site URL. Since we are in dev, we can use localhost. Save and Continue. There is some very useful info on the API, which is worth a read. Now under Settings, Basic, there is an App ID, which is your client id and the client secret. You can click Show to see it. We now have a client id and client secret. For Google, you can register with Google API. I go through how to do that in my previous course, Spring Security Authentication and Authorization in the module for OAuth. So check that out if you want to do that. Now this is an important point. Keep your secret safe! If it gets leaked, a malicious site could masquerade as yours potentially gaining access to your users' Facebook resources and your site. So now that we have the client ID and secret, let's configure the application next.

Sign-in with Google and Facebook

Because we are using Spring Boot in conjunction with Spring Security, configuration is straightforward. We add the spring-boot-starter dependency and, as our application is the client, the spring-security-oauth2-client dependency. Next, in the application's configuration file, application.yml, security, oauth2, client, registration, facebook as the application ID, and we add the client-id and secret. One final step. In the SecurityConfiguration class that extends the WebSecurityConfigurationAdapter, we need to add the OAuth2 builder method. For our demo application, this class already exists as we are using Spring Security in the application already with form login. Now if we refresh the page, we get a bland default page with a Facebook link, which we can select to initiate the OAuth2 flow with Facebook. Easy. Now to add Google, we need to add one other dependency, spring-security-oauth2-jose. This is for JWT support as Google uses OpenID Connect. So the ID token will be a JWT. For signing and encryption, JWT uses a collection of specifications. JSON Web Algorithms defines the list of algorithms for digitally signing or encryption. JSON Web Key defines how a cryptographic key or sets of keys are represented. Now this collection is known as JavaScript, object signing, and encryption. Now for some reason it's pronounced as jose rather than jose. Spring will use this library to verify the JWTs. Now the reason for these standards is that it makes life a lot simpler. If you are using a particular algorithm to sign your token and you describe it with the JSON Web Algorithms specification, then anybody reading that will know exactly what algorithm that is. Then if you specify the signature in the JSON Web Signature format, anyone can then read and verify that token or if you're encrypting if you used JSON Web Algorithms. And then if you provided the public keys that you used to sign in the JSON Web Key format, anybody can then verify your JWT automatically. So there basically is a group of standards that can allow anybody to verify your JWTs. Next, we add the client-id and secret for Google. That's it. Easy peasy. We can now login with Google. By default, Spring Boot allows us to also configure Okta GitHub in a similar way by just configuring the client-id and secret. Pretty neat, right? All that by adding just a bit of config. If you're like me, you're probably wondering what's going on under the covers? How is this configured? What decisions were made? You will learn this in the remainder of this module as it's important to know as no two security requirements are the same, and you'll probably need to configure and customize the framework. In our demo application, Spring no longer defaults to our original form login page. We will want to change this back to our original login page and add Facebook and Google sign-in links to it. Also, we have a bit of a hybrid implementation with form and OAuth2 login. We need a way to automatically register OAuth2 users, record their username and email, and set up and maintain their portfolio state. Let's do this next. But first, let's see what Spring Boot did under the covers.

A peek under the covers at the Architecture

Under the covers by adding the starter dependencies, all the required Spring Security dependencies for OAuth2 client are sourced for you. So you don't have to go hunt them down yourself. Because this is an MVC web application, Spring Boot plugged in a servlet filter, the delegating filter chain proxy, into the embedded Tomcat web server. If your typical Spring MVC application, requests are routed to the dispatcher servlet, which in turn routes them to the appropriate servlets to be actioned. In order to secure these requests, Spring Boot adds a Spring Security filter, the delegating filter chain proxy. This sits in front of the servlets and intercepts all requests to perform the security tasks and checks before allowing them to be actioned by the servlets. The delegated filter chain proxy delegates the request to a FilterChainProxy, which in turn assigns them to the appropriate SecurityFilterChain. Now filter chains are simply collections of filters that perform the security tasks and checks on the requests. You could have multiple filter chains in your application that specialize in dealing with different requests. Example, you could require that all requests to the admin URL use digest authentication while all other requests use basic authentication. Hence, you would configure two filter chains, one with a DigestAuthenticationFilter with a request matcher for the admin URL and another filter chain with a BasicAuthenticationFilter and a request matcher for all the other URLs. As the requests come in, the delegating filter will route them to the appropriate filter chain for the security checks. And only once the request passes the security checks is it allowed to proceed to the dispatcher servlet. To see this in action, let's put a breakpoint into the FilterChainProxy and see the filter chains that Spring Boot configured. Now there are three filter chains. The first two we can just ignore. They just catch requests to the /css, the /webjars URL and simply do nothing. It's because in our existing demo app, Configuration, we instructed Spring Security to ignore those paths as, by default, Spring Security blocks everything, and we want our unsecured pages like the registration and login to also be styled and have JavaScript even though the user's not authenticated. Now the third filter chain handles all our configured authentication and authorization. It has three key filters, the username and password authentication filter because our application still has its own form login configured and both the OAuth2LoginAuthenticationFilter and OAuth2AuthorizationRequestRedirectFilter. Now these two handle the OAuth2 authorization code grant flow. If we use our select sign-in with an OAuth2 provider, like Google or Facebook, the browser will send the request to the OAuth2 authorization URI with the provider's registrationId as a path variable. In our case, it would be in either Facebook or Google. This request will then be intercepted by the OAuth2AuthorizationRequestRedirectFilter, which will generate the redirect URI for the authorization server, which would include the CLIENT_ID, the requested scopes, in this case openid, to indicate we want the OpenID Connect flow, the response_type, indicating we want the authorization code grants, and optionally a unique state nonce. It will also cache this request in the user's session for future validations once Google and Facebook respond back to check the state nonce. Now the redirect requests will be sent via the user's browser. This is known as a front channel request. The browser will then forward it to the authorization provider, in this case either Google or Facebook. The provider can now look up the client ID, authenticate the user, and ask if they consent to provide the client with the requested scopes. If the user agrees, the provider sends a redirect request to the user's browser using the redirect URI provided in the initial request from the client, this time to the login/oauth2/code endpoint on the client. The authorization server will include an authorization code and the state nonce. Now on the client, this time the OAuth2LoginAuthenticationFilter will intercept the request. It will first validate the request by retrieving the state nonce from the session and comparing it against the one in the request. This is to protect against cross-site request forgery, then delegate to the AuthenticationManager to perform the actual authentication. The manager will consult with its authentication providers. Now there are many flavors of authentication providers depending on the grant type or if it's OpenID Connect or OAuth2. Now for the authorization code grant, the provider first exchanges the authorization code for the access token and optionally a refresh token from the token endpoint on the authentication server. You will also need to include the client-id and secret, by default, in the basic authentication header. If the OpenID Connect scope was included, then it will also retrieve an ID token. It will then validate the token returned. Now once validated, it needs to call the userinfo endpoint to retrieve all the resource owner's claims. To do this, it uses a user service, which come in two flavors, an OAuth2 version and an OpenID Connect version. The user service uses the access token against the userinfo endpoint, retrieves the claims, and packages them up into an OAuth2 or OpenID Connect user object, returning that back to the provider. The authentication provider then maps the scopes into granted authorities and packages up the user object, granted authorities, etc., into an authentication object of type OAuth2LoginAuthenticationToken, returning that back to the AuthenticationManager, which then forwards it back to the OAuth2LoginAuthenticationFilter. The filter will add the authentication to the security context held in the SecurityContextHolder. The holders is a thread local, so you can access it anywhere in your code to achieve the authenticated principal. Now additionally, the filter will create an OAuth2 authorization client object, which encapsulates the client metadata, like the token endpoint, userinfo endpoint, client-secret, and the validated tokens, like the access, refresh, or ID token, and then stores this object in the OAuth2AuthorizedClientService, which you can then wire into any of your classes if you need to retrieve the authorized client in case you need any of the tokens. Perhaps you want to use the refresh token to get a new access token or ID token. Now if we drill into the filters in debug mode, we can browse for all the components set up for us by Spring Boot, like the OAuth2LoginAuthenticationFilter. You can see its AuthenticationManager, and you can see it has the OpenID and OAuth2 providers user services. And all these were set up with just a few lines of configuration. Next, we will see how Spring Boot configured everything and its decision- making process as you will need to customize and override these decisions to tailor the framework to your application's unique security requirements.

Spring Boot auto-configuration of Oauth2

Just by adding the OAuth2 dependencies, the client-secret ID, everything was set up for us by Spring Boot. Now Spring Boot takes an opinionated view of what you want to configure based on certain environmental factors like what you added to the class path or what you had configured in your application context amongst other things. Spring Boot does this via these autoconfiguration classes. In the Spring Boot autoconfigure package, the Spring factory is located in the META- INF folder. You can see all the autoconfig classes that are bootstrapped. There are a number of Spring Security and OAuth2 autoconfiguration classes. The SecurityAutoConfiguration plugged in the delegating filter proxy into the embedded Tomcat web server. OAuth2ClientAutoconfiguration is what set up our application as an OAuth2 client. Firstly, it defines the order, stating it needs to be configured before the SecurityAutoConfiguration class. ConditionalOnClass states that the following classes need to be in application class path to activate this configuration. The final one checks if it is a web app. Basically, because our application is a web app and the ClientRegistration class is in the OAuth2 client dependency we just added, Spring Boot triggered this autoconfiguration as it assumed that's what we wanted. If this, however, wasn't the intention, we can let Spring Boot know not to configure this autoconfiguration by adding the exclude to your Spring Boot application annotation. Now back to the config class. Once the conditions are satisfied, two config classes are imported. These do the actual work. OAuth2ClientRegistrationRespository Configuration activated when we registered our clients in the properties files. It sets up the ClientRegistrationRepository, which manages access to all the client registrations. A client registration contains all the metadata about a client registered with a provider such as their registration and clientId, the clientSecret, the scopes it will send to the authorization endpoint, the authorization grant this client will perform, ProviderDetails like the authorization endpoint URI, userinfo endpoint, token URI amongst other things, everything needed to initiate the OAuth2 or OpenID Connect flow and to authorize the client. If you recall, we only have to provide the client-id and secret to configure sign-in with Google and Facebook. How about the other data? Well that's retrieved from Spring Boot's CommonOAuth2Provider enum. You can see the details for Google, Facebook, Okta, and GitHub, all the default config needed like the scopes. For Google, we have OpenID as its OpenID Connect provider. For Facebook, since it's not an OpenID Connect provider, we have the Facebook Connect scopes and all the nuance configuration for each provider, like the authorizationUri where the client can request the access code, the tokenUri where the client can exchange the access code for the access token, and optionally a refresh token, the JW keyset URI where our clients can get the public keys from providers to verify the signature of the tokens, userinfo endpoint where the client can use the access token to get more information about the user. UserNameAttributeName indicates which claim returned uniquely identify the principal and can be used for the username. Now you can see the benefits of using OpenID Connect. For the

OpenID Connect providers, the userinfo endpoint is always the same, /userinfo, and the username claim is always the subject. Whereas for each of the OpenID Connect providers, it could vary. You would need to check the provider specification. Now for Facebook and Git, Spring have done this for us, and it's the id claim. Now you can override any of these attributes in your properties files. If we change the scope for Google, you can now see our authorization redirect only includes the OpenID Connect scope. You can change the client-name, which will change the display name of the provider in the default login page. And you can change the registration ID by adding the property provider. The OAuth2ClientRegistrationRepository Configuration class, by default, creates an in-memory client registration repository and adds the client registrations to it. A client registration repository is just that. It's a repository for client registrations. It has one method, findyByRegistrationId, which returns a ClientRegistration. Now in your application, you might want to store the ClientRegistration metadata somewhere else like in a database perhaps. Hence, you can provide your own implementation of the client registration repository and add it to the application context. And as you can see, because of the ConditionalOnMissingBean annotation, Spring Boot would then back off. Now if we didn't provide these properties, we would have to programmatically expose the ClientRegistrationRepository as a bean. The other configuration class, OAuth2WebSecurityDuration, completes the setup. First, it configures an OAuth2AuthorizedClientService. Now on the successful completion of an OAuth2 authorization grant, the client is now authorized to make the call to the protected resource on behalf of the resource owner. Essentially it's an authorized client. Spring provides an OAuth2AuthorizedClient class, which is an association class of all the entities required to make the protected resource call. It has the access token, optionally the refresh token, which you can then use to get a new access token. There is a principle name, which is the unique identifier of the resource owner, the client registration, which has the provider's details like the token URI, the userinfo URI, etc., everything you would need to perform the protected resource call. Now the OAuth2AuthorizedClientService manages the OAuth2AuthorizedClients. It is a method to store, retrieve, and remove them. By default, Spring Boot configures an InMemoryOAuth2AuthorizedClientService. It also configures an OAuth2AuthorizedClientRepository. By default, it uses the session to store the authorized clients. Hence, once the user's session expires, you will lose the tokens. Now this could be an issue if you're using long-lived tokens, although best practice is to always keep the tokens short-lived to limit the impact if they get leaked. However, if you're using refresh tokens to keep the user logged in even after the session expires, then you could implement your own OAuth2AuthorizedClientRepository and store the refresh tokens in a different store. All you would need to do is create an implementation of the OAuth2AuthorizedClientRepository and register it as a bean with Spring. Spring Boot would then back off and plug your repository into the OAuth2AuthorizedClientService. And finally, the OAuth2WebSecurityConfiguration also configures the WebSecurityConfigurationAdapter and enables OAuth. In this case, we already have an existing adapter, so the ConditionalOnMissingBean would not trigger this default implementation.

Oauth2 login page

Since we added OAuth2 to our project, if the user is not authenticated, they are redirected to the Spring's custom OAuth2 login page, which is just a list of links to the providers we configured. However, we want it to default back to our custom login page and add the links to sign in with Google and Facebook to that. In our WebSecurityConfigurationAdapter on the OAuth2Login builder, we add our loginPage URI. We now need to add two links to our login page, starting with oauth2/authorization. If you recall, that's the URI the OAuth2AuthorizationRequestRedirectFilter will intercept. We add the registration ID for the provider, in this case Facebook and Google, and that's it. The user now has these options available. And if we select them, the Auth2AuthorizationRequestRedirectFilter will intercept the request and initiate the OAuth2 flow by sending the initial redirect to the provider, requesting the access code. Now if this is a new user, we need to register them by storing their name, email, and setting up a portfolio so that we can maintain the state of their crypto holdings each time they log in. We want to do this automatically without the user having to do anything. For them, ideally, it should be just sign in with Google or Facebook, and they're ready to go. Let's set this up next.

Automaticaly registering users: AuthenticationSuccessHandler

After a user successfully logs in via OAuth for the first time, we need to register them with the site and create a portfolio. To do this, we can use an AuthenticationSuccess handler. Simply create a class the implements the AuthenticationSuccessHandler and overrides the onSuccess method, giving us access to the request, response, and authentication objects. Next, in our configuration adapter, we plug in the handler via the oauth2Login method. Now our class will be called each time the user successfully logs in via OAuth. We need to check if the user already has a portfolio. Now in the current version of the application, users are uniquely identified by the username they provided when registering with the site. Hence, we need a unique identifier to use as the username. For this, we can use the value returned by the getName method on the authentication object and look up if there is a portfolio for that user. What is returned by this method depends on how the user was authenticated. If they were authenticated via the form login, then it will be the username. If by OpenID Connect, it will be the claim subject. And for non-OpenID providers, it can vary depending on the custom implementation. In the case of Facebook and GitHub, it's the claim ID. This can be a bit confusing at first. Basically, the authentication class implements the Java authentication and authorization service, Principal interface, and overrides the getName method. In the AbstractAuthenticationToken class, you can see this class returns a value depending on what type the principle object is. To better understand this, let's take a step back.

Retrieving claims form the Authenticated Principal

In the context of security, you have a subject, which is an entity. It can be a user like Victoria, a machine, another service, or a piece of code. In OAuth, the subject is the resource owner, hence why in OpenID Connect, the identity token has the subject property that uniquely identifies the resource owner. A resource owner might have the following attributes, username, email, social security number, name, address. Each of these or a combination could uniquely identify the user and our principals. Principals are subsets of a subject's attributes, and a subject can have many different types of principals. So for our subject, which is the resource owner, we can now access their principal via the getPrincipal method, which is the identity claim they approved for the client to retrieve from the identity server. If we now put a breakpoint and sign in with Google, you can see the authentication object is of type OAuth2AuthenticationToken. It's authenticated. It's for Google. The principal is the default OpenID Connect user as Google is an OpenID Connect provider. And you have access to the attributes you would expect returned via OpenID Connect. There is one authority. By default, Spring will add the user role for authenticated principals. We will discuss how to customize authorities later. NameAttributeKey, which gives the name of the claim used for the username as expected for OpenID Connect, it's the subject. And you can also access the ID token. If we do the same for Facebook, the principal is of type DefaultOauth2User as Facebook is not an OpenID Connect provider. NameAttributeKey is id, which is Facebook's unique identifier for a user. And we have a few attributes, email and name. As expected, there is no ID token as Facebook is not an OpenID Connect provider. Now the attributes you get back will vary between each provider and will depend on a number of factors. For Facebook, the user's public profile is configurable. They can decide what they want to expose. Hence, you get a subset of the following back. In my case, I only allow my name to be viewed. By default, Spring also requests the email scope. Facebook Connect also offers other scopes, say you wanted the user's location. Now in our configuration file, we can override the scope property for Facebook, add public_profile, email user_location. Facebook will now automatically detect the new scope and ask the resource owner for consent. You can see a warning here that some of the permissions below have not been approved for use by Facebook. So even if the user was to consent, Facebook will still not return their location as the client application is not approved by Facebook to handle anything other than public profile and email. We would need to request permission from Facebook to handle this type of user info and go through an approval process. That's understandable. Due to the recent scrutiny, social media providers are being a lot more restrictive now as you can imagine with all the recent public spotlight. The token would be of type OAuth2AuthenticationToken. To register a new user, we need the firstname, lastname, and email and a unique identifier. We can use the getAuthorizedClientRegistrationId to determine which provider the user is signing in with. If it is Facebook, we retrieve the email and name, which can be split to get a first and last name. For Google, since it's an OpenID Connect provider, the standards in OpenID Connect make this a lot simpler. We can check if it is an OpenID Connect provider by checking the type of the principal is DefaultOidcUser. Then we can use the standard OpenID Connect names for email, given_name, and family_name. We can then register the new user and create them a portfolio. Now the cool thing about the default OpenID Connect user object is that it provides the standard claims as accesses, which is quite neat. You can now log in with Facebook and Google. You can see the application displays the logged in the username in the top-right corner. With OAuth providers, this can generally be anything. For Facebook and Google, it's a long, unique identification number. So you might want to change that to display something more meaningful, like the name of the user. I'll show you how to do that later. As mentioned previously, Spring will add the default user role into the granted authorities. Next, we'll look at how to customize that and add some additional roles.

Mapping claims to authorities: GrantedAuthoritiesMapper

In Spring, there are roles and authorities. There is actually not much difference between them, and they are stored in the same granted authorities collection. What differentiates them is that roles are prefixed with ROLE_, and authorities are not prefixed. Granted authorities are used for authorization, like here where we specified that these two roles are required to access this URI. The hasRole would simply look up an authority that has the value role_user or role_admin. Whereas hasAuthority would look up the exact match. Conceptually, roles are used for coarse-grain authorization. You might have a user role so if an authenticated principal is assigned this role, they have comment permissions to access the site that, for example, a user would. Whereas authorities can be more finer-grained. So you might have some users with additional authorities to access premium content or special areas. Now by default, Spring will set one role of user for authenticated principals. But you may need to map some additional claims returned by the userinfo endpoint into authorities. And to do this, we can use the GrantedAuthoritiesMapper by creating a class that implements this interface and overriding the MapAuthorities method. Now at first it can be a little confusing. Why would the method have an authorities collection object to map to authorities? Aren't we mapping claims to authorities? Yes. The GrantedAuthorities mapper is called after the OAuth2 user object is retrieved by the user service. By default, the user service will add the role user and also scopes to the granted authorities. The mapper allows you to adjust the authorities after the fact. Now because the authentication process is not yet complete, there won't be an authentication object in the security context.

And to give you access to the token and claims, the user service adds either an OidcUserAuthority or an OAuth2UserAuthority to the granted authorities, which you can then retrieve and use to map claims to authorities.

The principal problem

Because our demo application has three ways to authenticate, form login, OAuth2 for Facebook, and OpenID Connect for Google, the getPrincipal will return three different types of objects. For form login it will be the type UserDetails. For OAuth2, DefaultOAuth2User. And for OpenID Connect, DefaultOidcUser. Now the OpenID Connect user extends the DefaultOAuth2User class and provides additional accesses like the idToken, which makes sense as the specification of OpenID Connect is built on top of OAuth. This can, however, make things difficult in your code. If we look at the authentication success handler, you can see the messy code, mainly because Facebook decided it wanted to be different and not be an OpenID Connect provider, which can cause your code to turn into a lot of messy if statements for each non- OpenID Connect provider. Imagine you wanted to get the name of user in one of our controllers. Now previously when the application just had form login, the UserDetailsService would extract the principal data from the database, create a user object of type MFAUser, return it to the authentication provider, which would then package it up into an authentication object. So the principal was always of type MFAUser. Hence in our controller, we could simply use the @AuthenticationPrincipal annotation and Spring would conveniently resolve it for us. Basically, it would be the equivalent of this. However, now if the user logs in with OAuth, this will simply return null. Let's say we wanted to get the name of the principal. We would need to do something like this. Yeah. First, we would check if the principal is of type MFAUser and extract the name. Next, we can assume they logged in via OAuth. We need to check if they are an OpenID Connect provider. If they are, great. We can use the standard claim names to get the first and last name. If not, then we would need to check which provider they are and surgically extract the attributes based on the provider specification. There has to be a better way. Next, we will look at how we can clean this up.

Customizing Oauth2 user types: CustomUserTypesOAuth2UserService

For non-OpenID Connect providers, the OAuth2User object is created by the OAuth2UserService and then packaged up as the principal in the authentication object. By default, Spring uses the default OAuth2UserService, which creates a default OAuth2User. Hence, we need an OAuth2UserService that can return a custom OAuth2 user, and it just so happens that Spring has an implementation we can use, the CustomUserTypesOAuth2UserService. Now let's first create our custom user type for Facebook, which is the only non-OpenID Connect provider at the moment. We override the getName authorities and attributes method. Next, we add the claims we want to extract. We saw previously Facebook provided the id, name, and email claims. So we can add these as fields. We add the accesses for these fields. The setters need to match the exact name of the claim. The CustomUserTypeService will then resolve them from what is returned by Facebook's userinfo endpoint response. For authorities, we will default to user for now, but you could add additional ones here based off the claims. We can now customize our attributes. Let's return the attributes Facebook provides. And in addition, let's also return the given name and family name in the same format as OpenID Connect. Okay, now in our configuration adapter, using the OAuth login builder, under the UserInfoEndpoint, we add our new class under customUserType for Facebook using its provider registration ID. That's it. Spring will now plug in the customUserTypes service into the OAuth2LoginAuthenticationProvider, and this provider will now return our custom OAuth2 user implementation for Facebook. Back at our OAuth2AuthenticationSuccess handler, after refactoring, we no longer have separate code to handle Facebook. We can also refactor our controller. This is cleaner, but we can do better. Let's create our own CryptoAuthenticatedPrincipal interface and the fields we care about, so FirstName, LastName, and FirstAndLastName combined. If we force our Facebook Connect user to implement it and also our MFAUser class, we now no longer have to cast between the two types in our AccountController. Now there is no custom user implementation for OpenID Connect as OpenID Connect is already a standard. So, to also force the OpenID Connect user object to implement our new interface, we have to create a custom OpenID Connect user service. So, let's do that next.

Customizing the Oauth2user with a Custom Oauth2userservice

The OAuth2UserService calls the userinfo endpoint to retrieve the user attributes and return the OAuth2User object. We want the OidcUserService to return a custom OAuth2User object that implements our crypto-authenticated principal interface. To do this, we can create a decorator to wrap the OAuth2 user and implement our interface. Next, we need a custom implementation of the OAuth2UserService to return our new OAuth2 user. We could implement the OAuth2UserService interface. But in this case, we will extend the OidcUserService and return our custom user. All we have to do now is to plug it into the userinfo endpoint in our configuration class using the OidcUserService builder method. That's it. Spring will now plug it into the OpenID Connect authentication provider. You could also do the same for a non- openIDConnect user service. We can now clean up our authentication success handler.

Module Summary

Users now have the option to sign in without providing their credentials using their Google or Facebook accounts. However, the application is still handling and storing user passwords and their personal details. In the next module, we will look at setting up our own authorization server and configuring our application to use it.

Delegating Authentication to an Authorization Server

Options for identity management

Now when it comes to authorization servers, you have a number of options. You could build your own. Spring provides a framework for you to do this. You could go for an out-of-the-box solution. There are many products from commercial to open source available. Or you can go with an identity as a service provider. Now there are pros and cons from each approach. And from my experience, an out-of-the-box solution or an identity provider as a service is probably the best way to go. Building an authorization server from the ground up is a lot of work, and it's not easy to get right. However, I will go over how you'd go about creating one with Spring Security and then show you an open source, out-of-the-box solution with Keycloak.

Spring Oauth2 Authorization Server

To create our authorization server, we create a new Spring Boot 2 project with web and Spring Security starter dependencies. We can use the Spring Initializer to set this up for us. If we look at the feature matrix, Spring's authorization server is not yet a part of Spring Security 5. It's a part of the older Spring Security OAuth project. Also, Spring Boot 2 only supports Spring Security 5, which will unify all the other OAuth projects eventually. Hence, authorization server should eventually be part of Spring Security 5. But until then, we can use the following dependency to bridge the old Spring Security OAuth with Spring Boot 2. We need to provide a version as it's an external dependency. The version will have to match the Spring Boot 2 version. Next, we'll create a configuration class for the authorization server that will extend the AuthorizationServerConfigurerAdapter. Then add the configuration annotation so that it's added to the Spring context. Additionally, we also add the EnableAuthorizationServer annotation. We override the configure method that accepts the AuthorizationServerSecurityConfigurer builder. Two things we need to do here. First, we set the permission for our tokenKeyAccess endpoint to permitAll, meaning any client can request that access token. In your implementation, you could restrict some clients from accessing the token endpoint based on roles or authorities. Perhaps the client first needs to be approved as not every grant type uses the token endpoint like the implicit grant type. Next, for the token check access, you can set it to isAuthenticated, meaning the client will have to be authenticated before it can use the token endpoint to verify a token they possess. If you're going to pass your token around to other services, then all the services would have to be known to the authorization server. We now need to register our web application as a client of the authorization server. We can override the configure method that accepts the ClientDetailsServiceConfigurer builder. For demo purposes, we'll set up an in- memory version. Here we can put all the details about our client web app, the client ID, supported grant types, in our case it's the authorization_code grant and the client-secret. Now it's important to keep the client secret safe. If that gets leaked, then anyone can masquerade as the client. Now for demo purposes, I have hardcoded it here. But in a production system, you would want to store it in a secure secret store. In my previous course on Spring Security, I talk about options of doing this in the Secure Secrets with Spring Cloud Vault module. Also, you would want your secret to be long and difficult to crack via brute force. And by default, Spring expects it to be hashed. So we can wire in a password encoder and do that. Next, we specify the supported scopes for this client and the redirect URIs. This is for validation purposes. If you recall, when the client requests an access code via the redirect from the user's browser, they provide a redirect URI in the request. The resource server will only redirect the URI that is preregistered for these claims. This is for the consent pop-up. By default, it's already actually false, meaning that the user will be prompted to approve the scopes being requested by the client. By setting it to true, the authorization server would just approve by default any scope defined in the supported scopes for this client. Now you can also store your client metadata in the database as Spring has a JDBC builder for that. Or you could create your own implementation by creating a class that implements the ClientDetailsService and implementing a loadClientById method that returns the ClientDetails object. Our resource server now needs a way to authenticate the resource owner, the user.

Authenticating the resource owner

If you recall back to the OAuth2 authorization code grant flow, the client sends the function or authorization request to the authorization server via our redirect from the user's browser. Our authorization server needs to authenticate the user and then ask them if they consent to this client request. Now to authenticate the user, it can ask them to provide a username and a password. So let's set this up. You can create a configuration class that extends the WebSecurityConfigurerAdapter. By now you're probably already very familiar with this class. Now when an instance of a configuration adapter is added to the Spring context, you're essentially creating a filter chain. Override the configure method that gives you access to the HttpSecurity builder. Because you can have multiple filter chains configured in Spring, in fact, by adding the

AuthorizationServerConfigurerAdapter in the previous clip, that created a filter chain. So now by adding the WebSecurityConfigurerAdapter, we will have two filter chains. We'll see these later. Now by using the requestMatcher, we can configure this filter chain to handle requests to the login page and to the oauth/authorize URI. Next, we need to configure how we want to authorize these requests. Now in this case, any request to this URI requires the user to be authenticated via form login. So what this means now is this filter chain will intercept authorization URI requests from the user's browser. It will then require the user to be authenticated. If they are not, it will prompt them to authenticate via form login. If the user is authenticated, the request will be allowed to proceed to the authorization endpoint. In order to authenticate the user, the authorization server needs to know about them, like the username, their password hash. And authentication in Spring, if you recall, is done by the AuthenticationManager. Hence, we can override the configure method that gives us access to an AuthenticationManager builder, which, as the name suggests, allows you to build an AuthenticationManager. The builder allows you to either create a JDBC version where it can retrieve the user's credentials from the database. However for now, we will create an in- memory authentication version and add a user Joe with a password. We'll give them one role, user. Now passwords in Spring Security need to be encoded. So let's return an instance of a BCrypt PasswordEncoder and use it to encode our password. Let's expose it as a bean as we also want to use this encoder in our authorization server configuration. This configures the AuthenticationManager in the adapter. We now need to expose it as a bean. We can do that by overriding the AuthenticationManagerBean method, which will return the configured version of the manager, which will then be used in the application throughout. Now once the user is authenticated and they consent, the request will hit the authorization endpoint, which will return an authorization code to the client. The client will then exchange it for an access token via the token endpoint. Then the client will want to use the access token against a userinfo endpoint to get the user-approved claims back. Now we need to create a rest endpoint for this. Create a rest controller, annotate the class with the RestController annotation, and let's create a UserInfo method and expose it via a Get method. For now we will just return a user object with the name, first and last name, and email all hardcoded. Now in your implementation, you can customize this any way you want. You can interrogate the authentication object to retrieve any approved scopes and selectively populate the response back to the client. Access to the userinfo endpoint requires a valid access token. So the authorization server also needs to play the role of a resource server. To do this, we add the resource server annotation. This will add a third filter chain that will look out for the access token in the header before allowing access to the userinfo endpoint. We will go over resource servers in more detail in later modules. Now because we're going to have free filter chains, we need to make sure that they're in the correct order. Now if we drill into the EnableAuthorizationServer annotation, you can see it has an order of 0. If we do the same for the resource server, it has an order of 3. And in the UserConfigurationAdapter, that has an order of 100, which would mean that the filter chain created by the AuthorizationServerConfiguration would be evaluated first, followed by the resource server and then the user configuration. Now in the OAuth flow, you wouldn't have an access token until the user is authenticated and they consent. So we need to allow requests to the authorize endpoint through. Now to address this, we could create a third configuration class that extends the ResourceServerConfigurerAdapter and configure the request matcher to either ignore requests to the authorize endpoint. However, changing the order is simpler. In the UserConfiguration class, set the order to 1 using the annotation. So it's before the resource server, which has an order of 3. This way the resource server filter chain is only evaluated if the OAuth2 flow was successful and the client has an access token. Now back to our application.yml config file, let's set the port number to 8081 so that it doesn't clash with our web app. Also, we need to provide a context-path. This is because we are running the demo on a localhost domain. Now a browser doesn't consider the port as part of a domain. So if you started the client web app and the authorization server together, the browser would store their cookies under the same domain, localhost, and that would cause a clash. The context root will ensure that the client web app cookies will be stored under localhost and the authorization server under localhost/auth. Next, let's configure our crypto web app to use this authorization server.

Outsouring user authentication to our custom autherization server

To register our authorization server with our client web app, in the application.yml file under security.oauth2 .client .registration, we use the provider property. We can give it a name, then provide the endpoint details of the authorization server, the authorization-uri where the client can redirect the user's browser to begin the OAuth2 flow and get the access token, the token-uri where the client can exchange the code for the access token, userinfo-uri where the client can use the access token to get the user's claims, user-name-attribute, what claim returned by the userinfo can be used to uniquely identify the user, like a username. Notice all the provider URLs contain the context path we configured in the auth server properties file. Now that the provider is configured, we need to provide the client metadata under security, oauth2, client. We give it a registration ID of crypto- portfolio, then the details we configured when registering it with the authorization server like the client-id, secret, client-name, a more descriptive name, which will appear in the default OAuth2 login page, scope, redirect-uri, grant-type, authentication-method, in this case basic. This is how the client will send the client-id and secret to the token endpoint when exchanging the authorization code for the access token. Now Spring also supports form authentication where the credentials are sent in the request body. And finally, the provider. We can now update our login.html page to add a link to login with crypto-portfolio and remove the previous login form as our application is no longer going to handle user credentials. Next, we'll take a peek under the covers to see all the moving parts and what made all this happen.

A peak under the covers of our Autherization Server

Let's talk through the authentication process in debug mode to better understand Spring's OAuth server and OAuth2 support. If we access the portfolio URL and hover over our new Login link, you see it's for oauth2/authorization/crypto-portfolio. If we click on it, the OAuth2 authorization request redirect filter will intercept the request. Retrieve all the client information we configured by looking up the registration ID and the provider used by the client with that registration ID. It will then create a redirect URI and send a redirect request to the client's browser. If we inspect the network traffic in developer mode in our browser, we can see the redirect request. It's to the authorization endpoint on the authorization server, auth/oauth, /authorize. It's requesting the authorization code grants, provides the client_id, the request scope and state nonce to protect against cross-site request forgery, and the redirect URI it wants the authorization server to respond back with the code. At the authorization server, let's put a breakpoint on the filter chain proxy. You can see three filter chains are configured. The first filter chain is intercepting requests to the authorization server endpoints, the token endpoint where the client can request the access token, the token key endpoint that will return the public keys used to sign the access token. This is so the client can validate the tokens offline. By default, it will be configured to deny all. We would need to set up the keys and open it up in the resource server config like we did for the other two endpoints if we wanted to use that. Then finally, the check_token endpoint where the client can send an access token and have it validated by the authorization server. Now this filter chain also has a basic authentication filter to authenticate the client. If you recall, the client can send the client-id and secret. Since this request is for the oauth/authorize, it will be intercepted by the second filter chain. If you recall, this is what we configured in the user config. This chain authenticates the user, i.e. the resource owner. You can see it has a UsernamePasswordAuthenticationFilter as we configured form login. The user will be prompted to log in. Once they log in, they will be prompted to consent the scopes requested by the client. Only then will the request be allowed to proceed to the authorization endpoint, which will validate the redirect URI against what we configured and send the redirect request to the user's browser with the code and state nonce. You can see this redirect request here. Our client path will intercept the request, validate the state matches the one in the original sent to the auth server to protect against cross-site request forgery and call the token endpoint to exchange the code for the access token, this time via our back channel directly to the authorization server via a REST call and not via the user's browser. The request will include the client-id, the client-secret, and targeted at the oauth/token URI, which will be intercepted this time by the first filter chain on the authorization server. Now you probably noticed the token endpoint is permitAll. The actual authorization endpoint also authenticates the client-id and secret. If everything checks out, it will return the access token to the client. The client's user service will then use the access token to call the userinfo endpoint. This time the request will be intercepted by the third filter, which is the resource server we configured. It has an OAuth2AuthenticationProcessingFilter, which will look out for the bearer token in the header to authenticate the request. Our web app will then use the attributes returned to populate the authenticated principal. We can now move all the authentication code out of the client web app and into our auth server. To the user, it will all look the same. If we actually disable consent, they won't even notice. But the client web app will no longer be handling passwords. If we look at the current state, it's a lot better. The authentication code is isolated from the application code. This is ideal as security changes are performed less frequently compared to application changes. And that reduces the risk of user credentials and data being leaked. Now further improvements that would need to be made is to move the Facebook and Google authentication onto the authorization server. That way our client app will only have to deal with one provider and reduces the risk of the other provider's secrets being leaked. It will also make adding additional providers transparent to our internal applications and performed in one place. You will also need to move user registration, email verification, implement password policy rules, security questions, two-factor authentication, the management of your secrets like keys, password hashing, to the authentication server. Now we are going to leave it at that as building an authorization server could be a course in itself. If, however, you're interested in adding these features to the authorization server, my previous course covers a lot of this. However, from my experience, it's simpler to just get an out-of-the-box solution. There are many available, some even open source, unless probably your organization already has one. We will look into one next.

Introducing Keycloak an out of the box solution for an Autherization Server

Keycloak describes itself as an open source identity management solution. In a nutshell, it's a Java-based authentication and authorization server. It has been around since 2013, has an Apache open source license, is actively being developed mainly by Red Hat developers, but also a large active open source community. There is also a commercial offering, Red Hat SSO, which is the same product, but it gives you that additional support from Red Hat that your organization might need. Now in my opinion, it has the best documentation and guides, which is one of the reasons I like to use it. It has all the features our application will require and more, single sign-on, supports all the key protocols, SAML, OAuth2, OpenID Connect, integrates with directory services so you can easily integrate it in your organization, and a multitude of features, like two-factor authentication, email verification, user registration, social sign-in, like Google and Facebook, and much more. It's also very customizable and extendable. You can write your own adapters and plugins and even format the login registration pages to your application's look and feel. Now I should mention, there is also CloudFoundry UAA Server, which is also open sourced and is actively developed by Pivotal, the creator of Spring and actually built with Spring. It's very similar and is also a good candidate for an authorization server. Next we will look at how to install Keycloak.

Installing and configuring Keycloak

You can download Keycloak from their web page. As of the creation of the course, I am using the latest version, which is 6.0 .1. Unzip it into a directory of your choosing. Then in the bin file, simply execute the standalone command with the jboss.http .port to 8081. Now the reason for the port is that, by default, it uses port 8080, and that would actually clash with our web app. If we access localhost on 8081, it will ask us to create the initial admin account. So fill that in. I'm just using admin and password for now. Now once logged in as admin, the first thing you need is a realm. This is like a container around users, applications, clients, etc. It's a core concept within Keycloak. Once your realm is created, you can start to see the power of Keycloak. With a few clicks, we can enable user registration, password recovery, remember me, email verification, SSL, select themes for our login page, configure security headers and content policies, enable user federation, like social sign-in with Google and Facebook, and much more. We need to register our application as a client. Under Clients, select Create. Give it a name. The protocol will be openid-connect. We can add a name and description. Make sure enabled is on. Consent required, this is if we want to prompt the resource owner to approve the scopes. Leave it as off for now unless you really want it. For the access type, select confidential, meaning a client secret is required and direct access grant to off as having it on would effectively enable the password grant flow. Service account enabled off. This is for the client credentials grant for machine-to-machine authentication. We don't need this at the moment. Below are some more advanced settings. a useful one is access token lifespan where you can set how long the access token is active for. Next, we preregister our redirect URI. This is the same one we provided in the Spring config. This needs to match as Keycloak will not redirect to any other URL. You could also add additional ones. Save this for now. In the Credentials tab, we now have the client ID and secret, which we will need to provide Spring later. Now that's it for the client. Let's create a test user. Select Users, Add user, fill this in and enable them. Add a password for them in credentials. As you can see, there is a lot of customization Keycloak allows you to do right out of the box. We are just scratching the surface. Now that's the hard part done. Next, let's set up this client in our Spring web app.

Outsourcing client Authentication to Keycloak

To set up the client we created in Keycloak, in the configuration file, we change the Crypto Portfolio client with the new settings. Add the client-secret. We can get that from the Keycloak admin console under the Credentials tab for that client. For scope, add openid as we will now be connecting via OpenID, and it has to be the first scope. Now if we go over to the Keycloak admin console under Client Scopes tab, you can see, by default, this client will always have the following scopes when generating the access token. But optionally, we can request the following. Now profile and email are default, and that's basically all we need for now. Next is the redirect-url. This needs to match the one we preregistered in Keycloak. Since the provider is now Keycloak, if we click the realm in the admin console, we will see a link to the OpenID Connect discovery document, which has everything we need to know about the provider or the endpoints, etc. Now we can provide all the configuration needed separately, like the JW keyset URI, the token URI, the userinfo URI. All these we can copy from the discovery document. But since we are using OpenID Connect, it'll provide us the issuer URI, and Spring will get everything it needs from the discovery document. We only provide the realm URI. Spring will know where to find the discovery document because the well-known openid-configuration URI is part of the standard. Now in the Keycloak admin, we can also add Google and Facebook as identity providers. Just copy and paste the client-id and secret and remove all that code to handle these. Now our client just cares about dealing with one provider, Keycloak. If we start our application, we can log in as Joe or via Google. This login page no longer serves a purpose. The application could just simply kick off the authorization process with the authorization server automatically. Now if you look at the index.html page, copy the authorize URI. Then in our configuration adapter for the client, set it as the login URI. Now when authentication is required, Spring will initiate the authentication flow with Keycloak automatically. Now Keycloak allows you to customize and style your login pages. So you can style it to your application's look and feel, and your users won't even know the difference. Now one final note. Keycloak also has adapters for Spring to make integration simpler. It has a Spring Boot startup project. But personally, I prefer to go with OpenID Connect. That way I'm not tied to Keycloak and can change whenever I want.

Introducing Identity as a Service (IDaaS) and module wrap up

Keycloak is great, but you still have to install it, maintain it, support it, keep it updated, and keep it secure. There is a third option where you can outsource your identity management completely to a third party. There are many out there to choose from. Two popular ones are Okta and Autho0. Their expert security teams look after everything for you. They support OpenID Connect, OAuth2, and they have a lot of guides to help you set up. So it's definitely worth considering. There is a course in the Pluralsight library, Build Your First Web App with Spring Boot and Angular, that has a module where it sets up and uses it as an identity provider. So check it out if you're interested. Let's recap what we have achieved so far. Our application is in a much better state. All the authentication functionality and code have been moved into the authorization server. Our application is now only an OpenID Connect client with one provider, Keycloak. Keycloak will now handle all the user registration, social sign-in, and the account management. And we can easily switch between any authentication provider. Now in the next module, we're going to take a look at how Spring Security OAuth2 supports public clients.

Oauth2 in a Client-side Single Page Application

The challenges for Oauth2 and public clients

If we recall back to the authorization code grants we used in our server-side web app, in the first part of the flow, the access code was retrieved by going through the user's browser, the front channel. Once the client received the access code, it exchanged it for the access token by going via the back channel, directly to the authorization server. The reason that worked is that the client can provide a secret to authenticate itself with the authorization server. And the reason for going via the back channel is that it is more simpler and more secure. It's just a direct call to the server via HTTP over TLS so it can't be tampered with. We can trust it because it came directly from the authorization server. Whereas via the front channel, we are actually using the user's address bar to transmit data between the client and the authorization server, which makes things more complicated and is not as secure as the back channel request. It's a little bit like the game of Chinese Whispers. Some malicious software on the user's browser or machine could tamper with the URL so you cannot completely trust it. The redirect is also more susceptible to man-in-the-middle attacks, and URLs are written to the browser history. However, there are some benefits to the front channel. And the reason OAuth uses the front channel for retrieving the code is that it is useless without the client secret, and it allows the user to be involved, to give consent. Functional requests also don't require the recipient to have an IP address, which works well with browsers and native web applications. Unfortunately, for public clients, making a back channel request is extremely challenging. Firstly, it is a cross-origin request. And until recently, browsers blocked these for JavaScript. But the main issue is storing the secret. Anyone can download the code for a JavaScript application and just view it. some malicious JavaScript could try to steal the token or the secret. And in the native app, it's the same issues. To work around these limitations, the implicit flow was created. It essentially is the same as the authorization code grant, but without the back channel request. Rather than returning the code back after the user consents, the access token is returned via the front channel to the public app. It's still a viable option, just not as secure as the authorization code grant due to going over the front channel. But to date, a lot of organizations still use it by ensuring they use correct content security policies and don't use questionable CDNs. Having said that, as of April this year, the OAuth working group no longer recommends the implicit flow for public clients, mainly because cross-site origin resource sharing is now universally adopted by most browsers. So the authorization code flow with proof of key exchange is now a viable method and is recommended. I won't go into the details of this as it could be a course in itself and is a little out of scope for the course. Either way, once you get the access token in your public client, you need to store it somewhere, which is another challenge for single page applications. Your options are either in the session, in the cookie where you will have to deal with cross-site request forgery, local storage where it will live forever unless your application specifically removes it, hence short expiration times is advisable in that case, and also session storage, which lives as long as your browser session as long as the tab in your browser is open. Additionally, it's not shareable between tabs, so it's slightly more secure than local storage. From the perspective of Spring Security however, your main concern is dealing with a request hitting your controller where the authentication took place externally. However, it's still worth understanding the challenges we just described so that you can add additional checks and scrutiny server-side. Now, for the remainder of this module, we will be working with a React version of the portfolio. And for the record, my specialty is not front-end development, so apologies beforehand for my JavaScript coding.

The new architecture of our SPA

The main difference is that the back end will now be a RestController rather than an MvcController as the communication between the JavaScript and the server will be done via JSON REST calls. It will be stateless and no cookies in the browser. We will have two microservices, one the portfolio and the other the support service. The React app will the client. And we'll set up an OpenID Connect flow using the implicit grant type between the clients, the user, and the authorization server. Once it gets an access token back from the auth server, it will include it in all requests to the microservices.

Configuring our public client in Keycloak

Because our client is now a public application, we need to configure a new public client with Keycloak. If we go to our admin console, create new clients, name it crypto-portfolio-react, and select public as the access and implicit flow. Authorization enabled off. For the redirect URL, we will now add the following for the React application. For web origins, we need to add our domain. This is for cross-origin requests. We will discuss what this is in a later module. Now because this is a public app, we don't have a client secret. That's it, we can just save it. Now in the React client, I'm using the OpenID Connect client JavaScript library. In the auth service, I set the provider URI, client_id, redirect_uri, scopes. And in my PortfolioActions class, all the calls to the microservices I include the access token. Again, this course is mainly focused on Spring, so I won't go over creating a React app in detail and how to secure it. That will be a separate course in itself. I believe there is a course on this in the library, which you can check out. There are also many different types of JavaScript frameworks out there. We will mainly focus on how Spring Security needs to be configured on the resource server side to work with a public JavaScript client. However, feel free to check the React code out and take a look. And I will over time be updating it in Git to improve the security, like adding the authorization code grant with PKCE as this is now what the OAuth2 group is recommending. At the moment, it's just using the implicit flow. Next, we will secure our microservices.

Securing the resource servers

The purpose of the resource server is to extract the JWT token from the request, validate it by checking the signature, the expiry, and optionally if the issuer and audience is correct, effectively to authorize the request to the protected resource. Currently, the two REST services are not secured, and anyone can access them. The main difference between the clients and these services is that the controllers are RestControllers and not the MvcControllers. Now this is not uncommon to find. The client and perhaps an API gateway would be accessible from outside a firewall and be secured, while all the services are behind the firewall and not secured. However, this is asking for trouble. Anyone behind the firewall can access the services, a rogue employee or a consultant, and there have been many high-profile cases in large organizations where the hacker was someone on the inside. Also, I've seen many cases where a configuration error resulted in a development or testing environment actually connecting to a production service, resulting in major production issues and impacting consumers. Converting the services into secure resource servers is very straightforward with Spring. We add the spring-boot-starter-security dependency and the spring-book-starter- oauth2-resource-server dependency. Spring Boot will now detect these dependencies and trigger the appropriate autoconfig class, securing all endpoints. The resource server will need a way to validate the token's signature. Hence, we even need to send a token to the authorization server for validation or have access to the public keys to verify the signature. In our application.yml file, we can add the JW key set URI or the issuer URI, and we can get these details from our Keycloak discovery document. Providing the issuer URI will allow for validating the issuer claimed in the token as an added security check. It will match the issuer in the discovery document with the one in the JWT token. That's it. If we try to access the microservice now, you can see the request is blocked as we are not authenticated. Easy. If we look at the filter chain, we can see that Spring has added the bearer token authentication filter, which, as the name suggests, will look out for the JWT token in the header, then delegate to the AuthenticationManager who will use a JwtAuthenticationProvider, which will decode and validate the token with the NimbusJwtDecoder and convert the trusted token into an authentication object using a JwtAuthenticationConverter. In later modules, we'll customize the decoder to add more validation to the token and the converter to custom map claims and scopes to roles. To recap, our React client will perform the implicit grants with Keycloak, get an access token, and then forward it in all subsequent requests to the resource server, which will then validate the token.

Retrieving claims of the Authenticated Principal

Previously when our services were not secure because they were stateless, the requester had to provide the username in the request either in the query string or the request body. That's not ideal as it opens up the application to injection attacks. Even though our services are now requiring a valid JWT token, it doesn't stop the holder of that token from accessing anyone's resources. You want to make sure that the holder of the token can only access their subject's resources. If we look at the JWT token created by the authorization server, it contains the users claim. Juan is the preferred username. So let's use this value rather than providing it in the request payload. In our controller, for the portfolio get method, we can access the JWT token authentication using the SecurityContextHolder. Then we get the token and retrieve the claim as a string. Now we don't have to pass in the username via query string or validate it. And we are confident that only the portfolio of the resource owner the token is created for is returned. We can actually make this cleaner by using the @AuthenticationPrincipal annotation as a parameter to the method. Spring will now do the retrieval and casting for us. Let's do the same for the other methods in the controller. Let's try it via our React app. Make sure you have Node running or the Crypto Portfolio React app running and both the support and portfolio services. I'm switching to Firefox as I find the developer mode shows the headers better than Chrome. I'm redirected to Keycloak. Hmm. Looks like there's an error, Cross-Origin Request Blocked. If I go to the Network tab, I can see a call via an option HTTP method, which returned a 401. Basically, Spring blocked this request. However, if I try the same request via Postman with the access token from the client app, it works and returns the data. This is because the request originated from JavaScript and was cross-origin. We will look at this next.

Cross-Origin Resource Sharing (CORS)

Cross-origin resource sharing is how browsers and resource servers protect users from malicious JavaScript. Say you're browsing a website. That website uses a third party ad company and loads ads that originate from different domains on the same page. The origin of the app's JavaScript is different from the main page. And if it was some malicious piece of JavaScript, you wouldn't want it to start making cross-domain requests. It could be trying to send your personal data out, such as items stored on your local storage, which could include cookies, tokens, session IDs, or try to make POST requests to your site. And these days sites get resources from all over the place, like CDNs, etc. In the past, browsers simply blocked all agent requests that were cross-domain, meaning only same-as-origin requests were allowed to address this issue. However, since then, JavaScript applications are more popular now, and the CORSs standard was introduced to provide a way to perform cross-domain requests safely. CORS is triggered any time a script makes a request to a different domain, a different subdomain, a different port, or a different protocol. For simple cross-domain requests like a GET, the server will still make the request to the resource server. The response will be returned, but the browser can block it from reaching the JavaScript unless the resource server returned a specific header approving the cross-domain request. For non-simple requests like PUT, DELETE, the browser can't just simply let them hit the server as they could perform an action, like remove crypto from your portfolio. So in order to check with the resource server if the request is allowed, the browser will do a pre-flight request to ask the server if the request is supported for that domain. If the response from the server contains the headers and support the request, only then would the browser make the actual request. This is why we saw the OPTIONS request in the Network tab. Okay, so next, let's see how Spring Security deals with these requests by informing the browser that requests from our React app origin are approved.

Enabling Cross Origin requests in Spring Security

If we look at the error in the console, it says the response from the pre-flight OPTIONS request was missing the Access- Control-Allow-Origins header. In the Network tab, we can see it received a 401 Access Denied. Basically, because we enabled Spring Security, all our endpoints are now secure and blocked for nonauthenticated users. So when the OPTIONS request hit the server, Spring Security simply blocked it. To allow pre-flight requests through, we need to configure a SecurityConfig class, simply a class that extends the WebSecurityConfigurationAdapter and override the configure method. Then with the builder, we add the CORS filter and configure all other requests must be authenticated and that we want a resource server with JWT enabled. Okay, Spring Security will now know how to handle cross-origin and pre-flight requests. However, we still have to instruct Spring what requests we want to enable for CORS. In our controller, we can do this at the method level using the CrossOrigin annotation. If we restart and try again, you can see the portfolio positions are displayed. If we look at the response from our resource server, Spring now included the Access-Control-Allow-Origin *. Basically, the * means it allows cross- domain requests from any origin. Now a lot of the time developers think great, that solved the error. But that's a bad way of thinking about CORS. It's not an error. It's another layer of defense to protect your users from malicious scripts. Hence, it's better to add our exact domain. Now the header also includes our domain, and the browser will ensure the origin matches. Otherwise, it will reject it. Additionally, you can configure it to specify what headers we support, what content type we support, and add multiple domains. So this gives you a lot of flexibility. Now all the other requests in the controller, like the POST, DELETE, are still blocked. We could add the annotation to each of the methods, or we could add it to the controller. Spring will then set the headers for all the below method types. You can also customize this annotation and specify what method you want, whether it's GET or PUT, but not DELETE, etc. Let's now check out the headers. This time let's add a position. You can see the browser sent a pre-flight OPTIONS request to the resource server, and Spring returned that POST is allowed and additionally that the authorization header is also allowed as our React app will be sending the token that way. The browser then made the actual POST request and received the response.

Module wrap-up and whats next

You now have the practical knowledge of how to apply OAuth2 and OpenID Connect to a basic monolithic MVC web application or a public single-page application, which communicates with a number of RESTful services. However, there are still some missing pieces to this puzzle. In the real world, you're more likely going to come across more complex security requirements where your MVC or SPA client communicates with multiple services where requests require multiple hops between these services to resolve. Or what if your resource server is not a human? How can you use OAuth2 for secure, machine-to-machine interactions? In the next module, we'll look at various security patterns you can add to your knowledge toolbox to deal with such scenarios when they come your way.

Oauth2 for Machine-to-machine Authorization

Module Introduction

If we inspect the current architecture of our web MVC version of the Crypto Portfolio application, it's effectively one big monolith, while our React client interacts with two microservices. In this module, let's fix and modernize the architecture. We'll break up the MVC monolith to also use the microservices and add another hop by adding a pricing service, which the portfolio service will use to source current crypto prices. This is common in many systems. You might have a mixture of clients, public like mobile and single-page applications that are running on the user's browser or on their machine. And we see applications server-side. So to prevent duplication of code, they will often interact with microservices. However, currently it is a bit messy. The clients have to be aware of where all the services are, and that can be tricky for public clients, like mobile apps, in case you need to change the location. Also, configuring cross-domain resource sharing in each service for single-page applications is also not ideal. Additionally, the clients might be lightweight, like mobile, and don't need the same large payloads returned back from the services. There might also be customizations required between each different client. Hence, you might have some sort of API gateway to route requests or use a backend for frontend pattern for public clients, effectively a service that sits between the client and the other services and routes and consolidates the requests for the client. Generally, that service is owned and maintained by the frontend development teams. From a security perspective, you now have more endpoints to secure and more hops to cater for. Effectively, a lot of the hops are now service-to-service with no user involved. Some are on behalf of the user and others are not. Next, let's look at security solutions for these challenges.

Security Challenges with Tokens in Distributed Systems

One solution is that once the client gets their job access token from the authentication server, the same token can be passed around between the services for authorization. It's called token relay just like in a relay race where the next runner starts when the baton is passed to them. This can work and like anything has both pros and cons. It's very simple to set up and configure. The token can have state, like the resource owner's claims and roles, which is useful for stateless services. But that can be both a positive and a negative. Because the token needs to be used across services, we have to pack it with all the claims, scopes, and roles that satisfy all services in the request chain. Hence, if it gets leaked in say the pricing service, the bearer can now use it against any service. Hence, the expiry needs to be short-lived to minimize that damage, which is a problem because the lifespan of the job needs to be enough to satisfy all the hops the service needs to make. If it expires mid-request, it will fail, and the client's exception handling will have to deal with that by either refreshing the token and resubmitting the request. Additionally, the pricing service doesn't need to know about the user's claims, which could contain things like their address, email, and any other identity information required by the other services. And if the developers of the pricing service decided to write the token to the logs, then over the course of a day, it could log a lot of tokens. And even if the tokens were expired, it could still be of use to a hacker if it contained claims like their address, email, and other sensitive information. Now between the single-page application and its BFF, it's totally a viable option. And if you're using OpenID Connect, you can even pass the ID token to it as really it's a part of the front end. With OAuth2 and OpenID Connect, if you're passing around the access token between services, another option to secure the user's data is to not put any user's claims on the actual access token, but require each service that needs the claims to make a call to the userinfo endpoint to retrieve them. The benefit of this approach is that services, like our pricing services, that don't need to handle any user claims no longer have to handle them as expired tokens are no longer of any use to a hacker if they are leaked as there's no sensitive data on them. So it just provides that little bit of extra added security. However, there is a performance hit with the additional hop to the userinfo endpoint. An alternative to token relay is for the services to get a new token from the authorization server to connect to the next microservice in the request chain. Let's look at that option next.

Introducing the Client Credentials Grant

In the client credentials grant, effectively no user is involved. The resource owner is the requester, most likely a non-human entity like another service. This is a great flow for when you have a batch job that needs to start up and make some requests to some services for reporting, etc., or in the case of our demo, the pricing service, which doesn't have any user resources on it. It's just the price of the crypto. So passing the access token is a bad idea for reasons described in the previous section. Also, the portfolio service might want to cache the prices and might not make the call to the pricing service for every request. It might use a thread that makes the request every 30 seconds, even though there is no active request taking place. With the client credentials grant, the client_id and secret is passed directly to the token endpoint in exchange for an access token only. No refresh token as the service can just rerequest a new token if the existing one expires. Now let's compare the pros and cons of the token relay versus client credentials and which part of the application they are suited for and also some alternatives.

Rethinking the Architecuture

Looking at our architecture, the client credential grant is appropriate for the requests between the portfolio and pricing service as there is no user resource on the pricing service. We could also use the client credentials for communication between all services. I've seen this approach used. It can work in an environment where every service trusts each other as there are pros and cons to consider. If the token gets leaked, it can only be used against the service it was intended for, which is great. However, since the resource owner is the other service and not a user, it won't have any user state on it. Hence, the user details need to be passed in the request body. So in case of the portfolio service or support service, the bearer can access any user resources. So they have to be very trustworthy as there is no way to verify that the user actually approved that request. Hence, you have to weigh up the pros and cons. For the demo, we will do token relay between the BFFs and the product and support service. Ideally, we should only include the username in the token or a GUID identifying them, which we can then use for authorization checks. All other details would be passed in the request body. And then those services will choose what they should and shouldn't pass down. Now there are some alternatives in the pipeline from the OAuth2 working group. One is Token Exchange and the other is JWT Bearer Token Grant. These will allow you to do things like impersonate a user. So for example, the web MVC app or the React BFF could exchange the access token received from the authentication server and approved by the user for another token that is a subset of the original token. Hence, if one of them was making a call to the portfolio service, it could use the current token and ask the authorization server for a new token with limited scopes and claims for what is required by the portfolio service. Hence, the bearer of that token cannot do anything on a support service. The bearer grant is not yet supported by Spring Security 5.1, but might be in Spring 5.2, and token exchange will probably follow. So look out for these. I will update the course once they are released. So if you're watching the course and they are made available, feel free to message me in the comments to remind me to get it added. Keycloak actually already has some support for token exchange.

WebClient vs. RestTemplate

If you have been using Spring for some time now, you're probably already familiar with the RestTemplate, which is used to make REST HTTP requests in Spring. However, if you look at its Java doc, they're basically saying it will eventually become deprecated and replaced by WebClient. And there is actually very limited OAuth support for the RestTemplate in Spring Security 5. The OAuth2 version of the RestTemplate is not a part of Spring Security as of 5.6 .1. Hence, you're probably better off choosing the WebClient over the RestTemplate going forward as WebClient can be used in both reactive and traditional web MVC applications. It supports both synchronous and asynchronous requests. By simply using the block, it will behave just like the RestTemplate. In the next sections, I will demonstrate how to use both to implement token relay and client credentials.

Token Relay with WebClient

Our React application will now connect to a single backend for frontend service, which will interact with the portfolio and support services via REST. Our MVC web application will also connect to the portfolio and support services. Let's start with the React public clients. Its backend for frontend service is a standard RESTful resource server. The React application will send the access token to it, the resource server will validate it using the keys from Keycloak, and then use the same token to make the request to the portfolio or support services. Now to do that, the BFF service will use a WebClient, and to set one up is simple. In a demo BFF service, we created a WebClientConfiguration class, which has a method annotated as a bean that returns an instance of a WebClient. You can use the WebClient class builder to create an instance and return that. Now you can autowire this WebClient instance to any of your classes. In the portfolio controller, we use the WebClient to make the request to the portfolio service. Here we are using the block method because WebClient is asynchronous. Block will force it to wait for the response. So it behaves just like a RestTemplate call. Now the portfolio service will require a valid token. So to include the access token in the request to the portfolio service, the WebClient has a convenient headers method, which you can use to add the authorization token. And it also has a convenient setBearerAuth method. All you need to do is pass in the access token. Now you can source the token from the authentication object, which you can retrieve either from the SecurityContext, which you can receive from the SecurityContextHolder. Or a cleaner and simpler approach is to use the AuthenticationPrincipal annotation. Spring will source and cast it for you. The JWT authentication object has an accessor to retrieve the token value, which you can then use in your header via the WebClient's headers method. Easy peasy, right? Now in our Spring MVC implementation, the only difference is the way we retrieve the access token as the MVC client is not a resource server, but a client. So it already has the access token stored in its authenticated client repository. Hence, we can use the RegisteredOAuth2AuthorizedClient annotation to retrieve the OAuth2AuthorizedClient object from this repository and then source the token from there. Now there are a few issues with this approach. Firstly, if the token expires before we make the request, the portfolio service will simply reject it. And because we're adding the token manually to the header, we would need to check the expiry of the token ourselves and then take the appropriate action before sending out the request. Next, we will look at an alternative way of including the token for outbound requests using the ExchangeFilter function that can address this issue.

ServletOAuth2AuthorizedClientExchangeFilterFunction

This time, let's switch over to our MVC web application. We create a WebClientConfig class and annotate it with the configuration annotation, then create a method that returns a WebClient, this time accepting two arguments, the ClientRegistrationRepository and the OAuth2AuthorizedClientRepository. Spring will autowire these instances for us. I'll describe what they're for shortly. We use the builder on the WebClient class and return the build method. We want the WebClient to automatically include the token along with our request in the authorization header. To do this, we can add an exchange filter, which will intercept the request before going out and add the authentication header for us. Spring provides an OAuth2 filter function just for this purpose, the ServletOAuth2AuthorizedClientExchange FilterFunction. Now that's a bit of a mouthful. To be honest, I had to practice that for some time, and it took me a few takes. But the name will make sense to you shortly. The filter function requires the clientRegistrationRepository and the OAuth2 authorizedClientRepository in the constructor. There are actually two flavors of the OAuth2AuthorizedClientExchangeFilter Function, the servlet and the server. Because this is an MVC application, we use the servlet version. If this was a reactive application, then we would use the

ServerOAuth2AuthorizedClientExchangeFilter Function. Now the ClientRegistrationRepository is used by the OAuthFilterFunction to retrieve the registered clients in our application. And in our application, it's the default in-memory version that loaded the client information from our configuration file. The filter function will need it to retrieve this information. Whereas the OAuth2AuthorizedClientRepository is a repository for authorized clients. Now an authorized client is just that, a client that has been authorized, i.e. it has a valid access token and optionally a refresh token. So basically the OAuth flow has happened for that client. The filter function will use the client registration to retrieve a client's details to make the OAuth2 authorization request to the authorization server and, once authorized, create an authorized client and store it in an authorized client repository where it can be access for future requests by the filter function to add the access token to the header until it expires. Now we plug in the OAuthFilterFunction into our WebClient. Okay, in our MvcController, we are using WebClient to send REST requests to our portfolio microservice. This time, we will let the ExchangeFilter function add the authorization header. Now an advantage of this approach is that if there is a refresh token and the client uses the authorization code grant, the filter function will check the expiry. And if it is within 1 minute of being expired, it will request a new fresh token. All the filter function needs to know is which authorized client we want to use as we could have multiple configured. One cool feature about the WebClient is you can add attributes to a request, which can then be referenced by an ExchangeFilterFunction. Here we can add the attribute for the OAuth2 filter function, the registration ID of the client so it can look it up in the authorizedClientRepository. There is a static method on the OAuth exchange function we can use to set the attribute it requires. So in the attributes property, we can use the clientRegistrationId, passing in crypto-portfolio. The OAuth2ExchangeFilterFunction will now look for that authorized client in the authorizedClientRepository. If it can't find one there, it then might attempt to authorize that client depending if the grant type is appropriate for that. Now we can also access the authorized client in our controller by adding the RegisteredOauth2AuthorizedClient annotation in our controller methods. Now Spring will resolve it. If we have multiple clients, then we also need to specify the registration ID so that it knows which one to retrieve. And we can use the OAuth2AuthorizedClient method and pass that in. Now let's test this. If we put in a breakpoint in the portfolio service BearerTokenAuthenticationFilter, we can see the token that was sent to the portfolio service by the MVC web app. You can also set the default client when configuring the ExchangeFilterFunction. Now one thing to note is in the OAuthFilterFunction, if the authorized client is using the authorize code grant, the filter will check the expiry each time. And by default, if it's within 1 minute of expiring, it will check for a refresh token for that client. If one exists, it will get a new token. Now that's quite neat. Our MVC web app is a client and can have access to a refresh token. However, if we pass the access token down to other services, we don't forward the refresh token. So all the other services that might be downstream of that request chain would not be able to refresh the token. Hence, 1 minute might not be enough. You might need to adjust the skew time based on the average time of your total request time. Token relay with the

Token Relay with RestTemplate

RestTemplate is also quite straightforward. When you configure your RestTemplate, you can add a ClientHttpRequestInterceptor. What this does is it intercepts all the requests before being sent out by the RestTemplate and gives you the chance to modify them before and after they're sent out. If we implement this interceptor, we get access to the request, body, and execution objects. Hence, we can get the access token from the authentication object, which is of type JwtAuthenticationToken and add it to the request header, then call the execute method, which will result in every request being sent out by this instance of the RestTemplate having the access token in the header. We just have to plug this interceptor into the RestTemplate. There is nothing more to do. However, there is no check to see if the token is about to expire and to refresh it. So bear that in mind. You might need to put that logic in there. We will discuss another version of the RestTemplate, the OAuth2 RestTemplate in the Client Credentials via RestTemplate section coming up, which can handle this for you.

Configuring Client Credentials in Keycloak

Our portfolio service needs to make a REST call to the new pricing service to retrieve the latest prices for cryptocurrencies in order to calculate the portfolio value. Since we won't be forwarding the access token to the pricing service for authorization, the portfolio service will need to source a new token from Keycloak using the client credentials grant. In the Keycloak admin console, we need a new client. Let's call it portfolio-service. Confidential as we need a client secret, and Service Accounts enabled ON. Everything OFF. Service Accounts ON enables the client credentials flow. Okay, that's it. Next, let's configure our portfolio service as a client to use this token.

Client Credentials with WebClient

In addition to being a resource service, the portfolio service will now also be a client as it will source a new token using the client credentials grant to access the pricing service. Hence, in the POM file of the portfolio service, we add the OAuth2ClientStarter dependency. Then in the configuration file, we add our client details. Add portfolio-service as the registration ID, client-id and secret, which we can get from the Credentials tab from the Keycloak admin console for that client. The grant type will be client_credentials. And for the provider, we will provide the token URI and add the provider to our client. When we configure our web client, we use the ServletOAuth2AuthorizedClientExchangeFilter Function again, passing in the ClientRegistrationRepository and the OAuth2AuthorizedClientRepository and adding it to the WebClient builder before building an instance of the WebClient that is exposed as a bean. Additionally, because this is the only client registered with the portfolio service, we can set it as the default by adding the registration ID. Basically, any time this web client is now used, the ExchangeFilterFunction will look up the OAuth2AuthorizedClientRepository for an authorized client with that ID. If one doesn't exist, it will look up the ClientRegistrationRepository to retrieve that client's information that we added via the properties file and contact the provider, which in this case is Keycloak, to retrieve an access token using the client credentials grant flow. Once the token is retrieved, it will be added to the authorized client repository for future access, and the token will be added to all outgoing requests via the WebClient. We can now use the WebClient to make the request to the pricing service. It just looks like any ordinary request with no requirement to add to header or attribute property. The ExchangeFilterFunction will now take care of the OAuth authorization side of things. One thing to note is that as of the current version of Spring Security 5.1 .6 that I'm using, once the OAuth2ExchangeFilterFunction authorizes the client, it will not refresh the token if it expires. The client credentials grant doesn't support the refresh token as that would not make sense. However, ideally, the ExchangeFilterFunction should check the expiry of the token before using it. And if it's expired, it should simply rerequest a new token using the client credentials grants. However, in Spring Security 5.1 versions, that doesn't happen. So when the token expires, it will keep using it in the requests, and you will eventually experience 401 exceptions as the pricing service will just reject the token. The Spring team have said that they will fix this in 5.2. So, if you're in the future and you're using 5.2 and higher, the ExchangeFilterFunction will check the expiry and get a new one for you. If you're using 5.1 like me, then you can either request Keycloak to have a longer expiry time. You can do that in the advanced settings. But that breaks the OAuth2 best practices. Tokens should have a short expiry time in case they are leaked to minimize any damage. Next, we will look at a workaround for this.

Client Credentials Token Refresh Workaround

So as a workaround for the OAuth2ExchangeFilterFunction in the client credentials grant not requesting a new token when the existing one expires, we can create a wrapper class around the OAuth2AuthorizedClientRepository. And in the wrapper, we can accept the actual OAuth2AuthorizedClientRepository as a constructor field. We implement the load, save, and removeAuthorizedClient methods. And for most, we simply delegate to the default implementation. But for the loadAuthorizeClient, we can check the expiry of the token. And if the token is valid for more than 1 minute, we can return it. Otherwise, we remove it from the repository. This will result in the exchange filter not receiving and authorized client back and hence will trigger it to request a new token from the authorization server. We then add our wrapper version to the ExchangeFilterFunction and that's it. We've implemented the client credentials flow using the WebClient.

Client Credentials Grant via RestTemplate

So as we mentioned before, Spring is moving to replace the RestTemplate with the WebClient. Unlike in the token relay example where the RestTemplate had to simply add the access token to the header for the client credentials flow, the RestTemplate also needs to request the access token from the authorization server using the client credentials grant before it can make the request. Now there is no support for that in Spring 5 as they are focusing on WebClient. So we have to use an older version of the Spring OAuth2 project, which has an OAuth2 RestTemplate implementation, which can handle this for us, meaning we have to rewrite our portfolio services security using the legacy OAuth2 project. In our project dependencies, we include the spring-boot-starter-security dependency and the following spring-security-oauth2-autoconfigure dependency to port it into Spring Boot 2.1 .6 as Spring Boot 2 doesn't support it out of the box. The version should be the same as the Spring Boot project. Next, we add our keyset URI as this IA resource server, and it needs a way to validate the token signature. We need to now create a configuration class and use the Configuration and EnableResourceServer annotation. We then return a bean of type ClientCredentialsResourceDetails where we put in our client details like the ID and secret, the grant type of client credentials and the access token URI, basically the endpoint where the RestTemplate can request the access token using the client-id and secret via the client credentials grant type. Then when we create an instance of the RestTemplate, we actually return an instance of the OAuth2RestTemplate with the ClientCredentialsResourceDetails object. We can then wire in the RestTemplate into our pricing service class and call the pricing microservice. The OAuth2RestTemplate will take care of the rest. It will get the access token from the token endpoint we configured and add it to the request header. Easy peasy. One minor change is required. The legacy version of Spring OAuth will expect a claim of user_name in the token, which it will then map to the username in the authenticated principal. So in Keycloak, we need a mapper that maps the preferred username to user_name. We can use the mapper type User Attribute to achieve that.

Module Wrap-up

In this module, we introduced the client- credentials grant and some key microservice security patterns that you could implement with Spring Security when dealing with different scenarios in distributed systems. Although JWT is great for storing state between service calls, which can improve the scalability of your application, you should still keep the content in them to a bare minimum, ideally a unique subject identifier and some scopes and roles. And unless your application really needs that scalability, sensitive data should be sent securely in the request body or sourced from the authorization server when required rather than being on the token as tokens can be passed around, and anybody who gets a hold of it can simply read it. If you do, however, need to include sensitive data on the token, then consider some form of encryption. Access tokens can be used by the bearer against the userinfo endpoint to retrieve the user's claims. Hence, be mindful of this when considering an access token for token relay. Now in the next module, we'll look at some vulnerabilities in our application and some advanced validation and customizations that you can implement to address these.

Enhancing with Customizations, Validation, and Exception Handling

Module Introduction

One of the characteristics of JWTs, OAuth2, and OpenID Connect is their flexibility. They're not as restrictive as other standards and very customizable. This can be a double-edged sword however with a positive and a negative and will mean that at times you will have to customize Spring's implementation to tailor it to your specific security requirements. And the Spring team have factored this in. In this module, you will learn how to modify the authorization request to the authorization server using a custom OAuth2AuthorizationRequestResolver. We'll also have some fun and identify security vulnerabilities in our application and why a valid token is not enough for authorization and how you can add additional custom token validation.

Customizing the authorization request

In an OAuth2 client, you might need to customize the authorization request that gets sent to the authorization server. You might need to add additional request parameters for the authorization code flow as providers might not always implement things correctly as per the specification or require custom parameters, which the standard does allow. Now by default, Spring uses the default OAuth2AuthorizationRequestResolver. If you recall, when you select the login, the request is for the OAuth2 authorization endpoint. Now the default OAuth2AuthorizationRequestResolver intercepts this, looks up the client via the registration ID in the client registration repository, and creates a redirect authorization request, which is sent via the user's browser to the authorization server. Now you might need to modify this, and Spring allows you to provide your own version of the OAuth2AuthorizationRequestResolver. Here I have an example of a custom implementation, which is a class that implements the OAuth2AuthorizationRequestResolver interface and overrides the required methods. This implementation is a wrapper around the default implementation. We simply create an internal reference of the default resolver, passing in the client registration repository. In the resolve method that returns the authorization request, we simply call the default implementation and then we can customize it before returning it. In this example, we can add additional parameters like the optional prompt, consent, which will prompt the user for consent each time they log in to the application. All we do now is add it, you guessed it, to the authorization endpoint via the OAuth2 login. Easy. Now if we restart our application, you can see the new parameter being passed in the authorization request.

Searching for security vulnerabilities

By default, the resource server will validate the signature, expiry, and if you provided an issuer URI, it will also check if the issuer matches. In our demo application, if you recall, the portfolio service requests a new access token using the client credentials grant to access the pricing service. Now the development team of the pricing service released some code that logged this token. If we were able to get access to these logs and try to use the token against the portfolio service, we'd get an error back. However, let's look more closely at this error. It's not an authorization error like a 401 Access Denied. It's an application error. The request actually got past Spring Security. The reason it failed was that the token doesn't have the user state, like the username, and the application actually requires that. However, we got through security. We could try other endpoints. In fact, the support service has some endpoints that allow the username to be passed in the query. Now the token, as per the Spring Security resource server, has a valid signature, is not expired, and because all our tokens are issued by the same issuer/realm in Keycloak, the issuer attribute on the JWT also matches. So as far as Spring Security's concerned, the request is authorized. Now this is a classic vulnerability. I've seen many of these time and time again when reviewing security implementations, particularly in distributed systems. It's often services, like the pricing service, that get neglected as developers think what's the worst that could happen? It only provides pricing. However, they can become backdoors into the system, especially when tokens are being passed around. I've even seen many cases where unsecured services were receiving tokens in the header from clients because the RestTemplate or the WebClient was configured to, by default, include the access token on all outbound requests and some new feature required a call to an unsecured service. The developer just wired in the WebClient or RestTemplate without realizing that, by default, it was also including the access token, hence why it's always preferred to include the token manually on all requests rather than doing things by default. Additionally, the exception handling is very poor in this service. It should return a generic error HTTP code and not a stack trace as this gives clues to the hacker. Now one way to prevent the pricing token being used against the portfolio or support service is to validate the audience claim on the JWT. Only tokens intended for the pricing and support service should be accepted by them. We will look at how to do this validation next.

Performing custom validation of the JWT

By default, the authorization server will use the JWT authentication provider, which will decode the JWT token using a JwtDecoder that will perform the validation of the token, checking its signature, optionally the issuer and the expiry. In our portfolio service, you can implement a custom validator to check the audience and add it to the JwtDecoder. To do this, create a class that implements the OAuth2TokenValidator and override the validate method. Here you have access to the JWT object where you can access the audience and check if it contains the value portfolio-service. If it doesn't, you can return an OAuth2Error, otherwise return success using the OAuth2TokenValidatorResult object. Now you need to plug in the validator into the decoder. To do this, create a new instance of the decoder. In the SecurityConfiguration class, you can create a method that returns an instance of a JwtDecoder. Here you can return an instance of the NimbusJwtDecoder as this is the default for Spring by using JwtDecoders builder. Since the issuer supports OpenID Connect discovery, you can use the issuer URI method and pass in the issuer URI. Next, you need to create an instance of the default validator so that the default checks like the expiry, the issuer claim are still performed. You can use the JwtValidator builder class for that. Now you can only set one validator into the decoder. So to add both the default and our custom audience validator, you need to create an instance of the delegating OAuth2 validator and pass in the both instances in the constructor, then add the delegatingValidator into the decoder and return it. And that's it. Plug this decoder into the JWT authentication provider. Now do the same for the support service, but this time checking the audience is support-service. One final thing that's needed now is to add these audiences into the access token created by Keycloak. For that, you need a mapping. So in our Keycloak admin console, in Clients, crypto-portfolio, Mappers, Create, give the mapper a name, and the type will be Audience. Here you can select from a list of clients Keycloak is aware of. Now our portfolio service is a part of this list, so you can select it. This is because we configured it as a client previously so that it can request a token via the client credentials grant to make a request to the pricing service. Make sure it is added to the access token. You can create a similar mapping for the support service. However, the support service is not a client, so it is not part of the client list. Hence, you can use the custom audience field and just simply type it in. That's it. The audience on the access tokens created by the WebClients will now contain the portfolio and support service. Whereas the token created by the client credentials grant by the portfolio service to access the pricing service will not contain these audiences and hence cannot be used against the portfolio or support services.

Final thoughts

The key takeaways from this module is that for authorization, it is not adequate to solely rely on the validity of the access token and whether it is expired or from the correct issuer. Distributed applications are as strong as their weakest link. Any service could become a backdoor into the application, so be vigilant. Now in addition to the authorization endpoint, there is also a token endpoint where you can add a custom access token response client to customize the outbound request to retrieve the access token from the token endpoint. Use the audience attribute to ensure the token is used in the proper context. Although the RFC doesn't provide much restrictions on what the audience field should be used for, it's important not to get too granular with your audience validation. It should be for who can receive and process the token, not what resource or actions the bearer of the token can perform on behalf of the resource owner. That's what scopes are for. As you have seen, you need to rely on multiple layers of defense. In the next module, we will look at more finer-grained authorization using scopes, roles, and authorities.

Layering Scoped-based Authorization

Module Introduction

We have an application secured with OAuth2 and OpenID Connect. Our users' personal data and credentials are now safely stored away from our application's code, and none of our application services handle any passwords. So at best, if a hacker was able to extract any portfolio data from our services, it would most likely be useless. The only identifying property they would have is the username. Now we could use a GUID identifier to make it even more anonymous. Now in any modern application, there are multiple layers of security. There's authentication, identifying and validating the subject and their claims. And then there's authorization, once authenticated what the subject can and can't do. If we go back to our train station analogy, if you have a valid train ticket, then you're authorized to ride the train as a passenger. But you're not authorized to drive the train. That's for a qualified train driver. Next, we will focus on providing more finer-grained authorization using scopes and authorities. But first, let's have some fun and poke some holes in our application's security.

Searching for More Security Vulnerabilities

Okay, let's put on our hacker hoodie and head over to a dark, dingy basement. If you look at the application's architecture, the portfolio service uses the client credentials grant to request a new token from the authorization server to access the pricing service. Now we don't have access to the client's secret, so we can't get the token using the client credentials grant. And if we access the service without a token, as expected, we get a 401. But in our React application, it gets an access token directly from the authorization server using the implicit flow. So a hacker can register a portfolio, and if they inspect the network communication, you can see the access token received back from the authorization server. If they include this token in their request to the pricing service, they now get a response. Effectively, all the pricing service is doing is checking if the token is valid, if it's signed by the authorization server. So any token issued by our own Keycloak is valid. Next, our support service uses a username parameter. Hence, I can use the same token and just substitute anyone's username to get their support queries or even brute force the values with a valid query ID. Often users can include sensitive information about their account in support queries. So a hacker would definitely be interested in browsing these. If the hacker gets a valid ID, they can even post a support query. Now with this security vulnerability, you could even post back impersonating an admin, asking the user what their password is. Now in the remainder of this section, let's plug the security holes in.

Scopes vs. Roles vs. Authorities

Now first it can be a bit confusing the difference between scopes and roles versus authorities. In OAuth2, scopes define the scope of the access the resource owner has approved the client to perform on their behalf, generally in the token they have defined under the scope or scp property. However, sometimes they can be claims about a resource owner like their roles, which also need to be considered. For example, a user, the resource owner, might have approved a client to access the administrator functions of the application on their behalf by including the scope admin. However, the user might not be an admin. So even though the resource server trusts the user approved the client to access the admin section by including the admin scope, it can still reject the request because the user is not entitled to access the admin section. They don't have an admin role. And vice versa, the user might be an administrator, i.e. having an admin role, and be entitled to access the admin section. However, they did not authorize the client with the admin scope. Hence, even though the resource server can verify the user is an administrator, it will still reject any access to the administrator resources for the client because they don't have the appropriate scope authorized by the resource owner to access the administrator resources on their behalf. Now Spring stores roles, scopes, and authorities in the same collection called the granted authorities. In order to avoid name collisions, say you have an existing application with the role of admin, you configure OAuth2 in your application and the scope of admin is in the JWT, both mean two different things. The role indicates the user is an administrator, and the scope indicates they authorize the bearer to access the admin resources on their behalf. So in order to have both a scope and a role with the same name, Spring uses the prefix ROLE_ for roles and SCOPE_ for scopes. But for authorities, there is no prefix. Hence, in the code if you see hasRole ADMIN, Spring will actually search for a value of ROLE_ADMIN in the granted authorities, which is the same as using hasAuthority ROLE_ADMIN. If you need to check for a scope, you can use the hasAuthority SCOPE_ADMIN.

Adding Scopes and Roles to Keycloak

Currently in our application, we have users Joe and Dave and a service account portfolio service. The portfolio service should be the only client with access to the pricing service. So let's create a new scope called pricing and only allow access to the pricing service to clients with that scope. In our Keycloak admin console under Client Scopes, select Create. Fill the form in. This will be for a non- human resource owner, so we don't need consent. Save. One more scope for administration. Call it portfolio-service-admin. Now to add the pricing scope to the token created by the portfolio service client, navigate to the client, select scopes. You can either add it as default, meaning it will always be added to the token for this client without them having to request it during the authorization or optional, meaning the client will have to specify that they want this scope in the authorization request. Let's put it in as optional. Let's do the same for the portfolio service admin scope, but added to the crypto- portfolio client. Hence, only users who authenticate via the MVC application will be allowed to access the admin features. If they log in via the React client, they won't have the appropriate scope even if the resource owner has the admin role. Next, let's create an admin role for the Crypto Portfolio clients. On the client page, Roles, Add Role. Name it portfolio-admin. Now to include the roles in the access token, we can create a mapper. Create new, token claim name to roles. This would be the key of the new attribute. The value will be a collection of roles. Now that the roles and scopes are configured, let's create an admin user, Victoria. Let's fill in the form and the credentials, give her a password, and assign her the portfolio-admin role. Okay, all set. Next, let's use these roles and scopes in our application.

Authorization at the URL

Let's configure our pricing service to only allow access to authorized clients with the pricing scope. If we look at how Spring Boot configured our resource server in the OAuth2ResourceServerAutoConfiguration class, which used the OAuth2ResourceServerWebSecurity Configuration, by default it's all requests should be authenticated by a resource server using JWT. Let's override it. To do that, we need to create our own config class that extends the WebSecurityConfigurerAdapter, you're probably already very familiar with this class, and override the configure method is the HttpSecurity builder. Configure it to authorize any requests that have the authority SCOPE_pricing. If you recall, all scopes are added to the granted authorities by Spring by default with a capital SCOPE_. And enable ResourceServer with JWT. Now only the tokens created by the portfolio service via the client credentials grant will have the pricing scope and therefore have access to the pricing service. In our portfolio client's MVC web app, let's go through some of the more complex URI authorizations. Here we are using an mvcMatcher to be more specific. For the login and login-error page, we use the permitALL, which will allow unauthenticated users access to these URLs so that they can authenticate themselves. Next, for /portfolio, which is used for the portfolio controller, only authenticated users with the role USER have access. For the support page, it's USERS and ADMINS. Now for the admin URL that maps to the admin controller, we want only clients with the approved scope of portfolio_ADMIN. So we use the hasAuthority, the SCOPE_ prefix, followed by a portfolio_ADMIN. In effect, our React clients will never be able to generate a token with the portfolio admin scope. And therefore, even users with the admin role will not be able to access admin features. If we look at the token that will be generated for Victoria via the Crypto Portfolio clients, the scopes will be automatically mapped by Spring into the granted authorities, but the roles won't be by default. We need to instruct Spring on how to map them to the granted authorities. So in the GrantedAuthoritiesMapper that we created in module 2, we simply look up the roles attribute. And if it contains our portfolio-admin role, we add an admin role to the granted authorities. Next, we need to do the same authorization in our portfolio service as we don't want someone to get around this authorization by going directly to the portfolio service. Let's look at how to do that next.

Mapping Roles and Scopes from Your Token into the Principal

In Spring Security OAuth, the authentication providers are responsible for validating the access token and converting it into an authentication. Now there are different flavors of authentication providers. On the client side, we have different flavors of providers depending on if you are using OAuth2 or OpenID Connect and what grant type was chosen. There you will use a GrantedAuthoritiesMapper to customize your granted authorities. Check out module 2, which goes over that. For our resource server, the default provider is the JwtAuthenticationProvider. If we look at its implementation, it first uses a JwtDecoder, which validates the token. You customized this in module 7 to add additional validation. Once the token is validated, it is converted into an authentication object with all the claims, scopes, and granted authorities. Hence, you need to customize the converter to add roles to the authentication principals authorities. The default implementation for the converter is the JwtAuthenticationConverter. You see it has the getScopes and extractAuthorities methods. For scopes, it uses the WELL_KNOWN_SCOPE_ATTRIBUTE_NAMES to find them in the JWT and adds them with the SCOPE_PREFIX. Hence, we can create a class that extends the JwtAuthenticationConverter and override the extractAuthorities method. Here we can call the parent to get the scopes. If you wanted to, you could get the scopes and map them yourself. You could get additional ones from a database or other location or even add them to the granted authorities without the SCOPE prefix. Now for our use case, we'll extract the roles claim from the JWT. And if the user has a portfolio-admin role, we will add an admin role to the granted authorities and return it. Now in our SecurityConfiguration class, we can just add it to the JWT endpoint, which will now plug in our version of the converter into the JWT authentication provider. Now the state of the user is coming from the JWT and not the database. Next, let's look at a more finer-grained authorization at the method level.

Securing Your Methods

Let's fix the security vulnerabilities in our support service. In the admin controller of the support service, we want only users with the admin role to access these features and clients with the scope portfolio-service-admin. We could do that by using the mvcMatcher in our WebSecurityConfigurationAdapter to apply this at the URL level. But we can also do the same in our admin controller by simply enabling method authorization using the EnableGlobalMethodSecurity annotation. We can use the preauthorize annotation on any of the REST methods to check for the role admin and scope. For scopes, we use the hasAuthority. For the admin role, we can use the hasRole ADMIN. For scopes, we can use the hasAuthority. And to combine them into one, we could use the hasAnyAuthority annotation. For the admin role, we need to include the ROLE_ prefix as that is how it is stored in the granted authorities. The hasRole method did the prefixing for us. For the scope, we use the SCOPE_ prefix. In our support query controller for the getQueries method, we can use the preauthorize to allow clients to perform actions on behalf of administrators who have consented to the admin scope to view anyone's support queries, while non-admin users can only view their own. We allow access if the principal has a role and scope of admin or if they have a role of USER, and the username path variable matches the preferred username in the JWT token claims. Basically, administrators can view all queries and respond to them. Users can only view their own and respond to those. If the admin is using the React UI, it will not include the admin scope, and hence they won't be able to view all the queries and perform administration tasks. There is also a PostAuthorize annotation where you can perform security checks prior to the method being returned so you can interrogate the objects being returned, modify them, remove things from collections, etc. You can see how powerful Spring expressions are. Next, we will look at how you can learn more about Spring expressions and authorization.

HTTPs and Further Learning Oportinitues

In this module, we only scratched the surface of the features Spring Security provides for authorization. Spring also supports even finer-grain authorization at the entity level where you can define access control to instances of your portfolio, like if you want to allow users to grant access to their portfolios to other users and friends. One important point. All communication in this demo is currently unsecured over HTTP, which means that anyone listening to network traffic can see the bearer tokens and simply use them. In a production system, you need to use TLS for all communication containing tokens and user data. Spring makes configuring this a breeze. In this module's demo pack, I will include a final version secured with HTTPS. If you're interested in learning more, check out the course Spring Security: Authentication/Authorization - Building Effective Layers of Defense, which covers how to configure HTTPS. Also, there is a module, which goes over more advanced authorization features in more detail.

Course Complete Whats Next

As you can see, it's important to use a defense in depth security strategy. It's like airport security check-in points. First, it's at the desk. They check your passport, issue you a ticket. Then you use the ticket to get through the first gate followed by customs security checks and passport checks. Once in the airport, other checks before you get on the plane to make sure you're boarding the correct flight and then inside the plane to make sure you're sitting at your designated sea. And even once you exit the flight, there are checks at the other end before you leave the airport, the same thing in our application. We use an authorization server with OAuth2 and OpenID Connect to authenticate the users and clients and then issue them a token. When the token is used, we validate the token is not fraudulent by checking the signature, expiry to ensure it's still valid, the issuer to ensure it was issued by the correct authorization server entity, the audience to ensure it is intended for the resource server. Once in the resource server, the scopes to ensure the client has received consent from the resource server, the roles to ensure that the resource owner is entitled to give the consent. And Spring provides the tools for us to perform these checks at multiple layers, at the URI, method, and event entity level on the way in and on the way out just like airport security. OAuth2 support in Spring Security is really building momentum. In version 5.2, they are rolling out opaque tokens, support for the JWT bearer grants followed by the token exchange. So I will be providing updates. Hence, subscribe if you're interested and want to keep up to date. And feel free to message me in the discussion if you have any questions. Now that concludes the module and the course. If you have stuck with the course from the beginning, well done. You will now have OAuth2, OpenID Connect, and JWT in your security skills toolbox. And from my experience, you're way ahead of most developers who for some reason find security a chore and avoid learning about it to the application and user's peril. Introduce a bug in the new feature, most likely you'll have to explain it to your tech lead or your manager. But introduce a security vulnerability, and you could be explaining it to very senior management, compliance, or even legal.