1 What are OAuth 2 and OpenID Connect?

This chapter covers

- Understand the purpose of access tokens
- Get how tokens are issued and validated in an OAuth 2 system
- Know the roles involved in an OAuth 2 / OpenID Connect system

Suppose you work in a large organization and have to use several tools for your daily work. You use bug tracker apps, apps for documenting your work, apps for registering your time, and so on. You need to authenticate in each one of these tools to work with them. Would you use different sets of credentials for these apps? Of course, using different sets of credentials could work, but this approach would be cumbersome for the user (you) and would also complicate the purpose of the apps you work with.

For you, the complexity comes from the fact that you'd have to remember the credentials and log in several times in each of the apps you use. For the apps, the added complexity comes from the fact that they'd also need to implement the capability of persisting and protecting the credentials and the actual authentication.

What about managing the responsibility for storing the credentials and authentication in a separate app? In this case, the users would have to only log in once, and use all their apps without bothering to authenticate repeatedly. Is there such a solution? Yes. You can implement authentication over the OAuth 2 specification. The second thing is that we can go even bigger. An app that can be used by public users (users outside of an organization – an app you create for the whole world) might also need authentication capabilities. The app can implement those capabilities, but:

- Implementing authentication in this app requires more work and effort

- The users need a specific set of credentials for this app
- The users sometimes don’t trust creating specific credentials for any small app they use

Could you maybe allow the users of this app to log in with some credentials they already have? Could your app’s users log in with their Facebook, GitHub, Twitter, Google account credentials instead, maybe? You might have already seen that all over the place. Apps around the web allow users to choose to sign up and use various social media platforms. This way, an app allows its users to authenticate with a set of credentials they already have without you needing to implement something specific in your app. Such an approach

- Reduces costs (of implementing and maintaining authentication in your app)
- Avoids user trust issues (of having to register and create another set of credentials that your app will maintain)
- Helps users minimize the number of credentials they use.

OAuth 2 is a specification that tells one how to separate the authentication responsibilities in a system. This way, multiple apps can use one other app that implements the authentication, helping the users to authenticate faster, keeping their details more secure, and minimizing the costs of implementation in the apps.

We’ll start with section 1.1, where I’ll introduce the main actors involved in a system where the authentication and authorization are built over the OAuth 2 specification. In section 1.1 you’ll learn all the responsibilities of the parts of an OAuth 2 system, such as the user, the client, the authorization server, and the resource server. In section 1.2, we discuss tokens. Tokens are like access keys for an app. You’ll learn you can use multiple types of tokens and when it's best to use each type. In section 1.3, I review the most critical ways tokens can be issued (which we’ll implement and test in chapter 2). We end this chapter with section 1.4, where we’ll go through possible pitfalls you need to consider when implementing OAuth 2.

Before we start, I’d like to mention that in this chapter, I offer you the simple perspective of everything you need to know to further understand our discussion in chapters 2 through 4. I don’t intend to make you an OAuth 2 and OpenID Connect expert in one chapter. I don’t think that’d be possible, since the two are quite complex enough for others to have written entire books about them. If you wish to extend your knowledge on the subject, I recommend *OAuth 2 in Action* by Justin Richer and Antonio Sanso (Manning, 2017) and *OpenID Connect in Action* by Prabath Siriwardena (Manning, 2023).

1.1 The big picture of OAuth 2 and OpenID Connect

Suppose you need to attend an interview with a large corporation. You have been called to their main headquarters for a face-to-face discussion. But not just anyone can enter the company’s office. There are specific protocols implemented for visitors.

To get in the building and attend the discussion, you must first visit the front desk and prove who you are by using an ID. After getting identified, you’ll get an access card from the front desk, allowing you to open certain doors. You might not even be able to use all the elevators but only specific ones (figure 1.1).

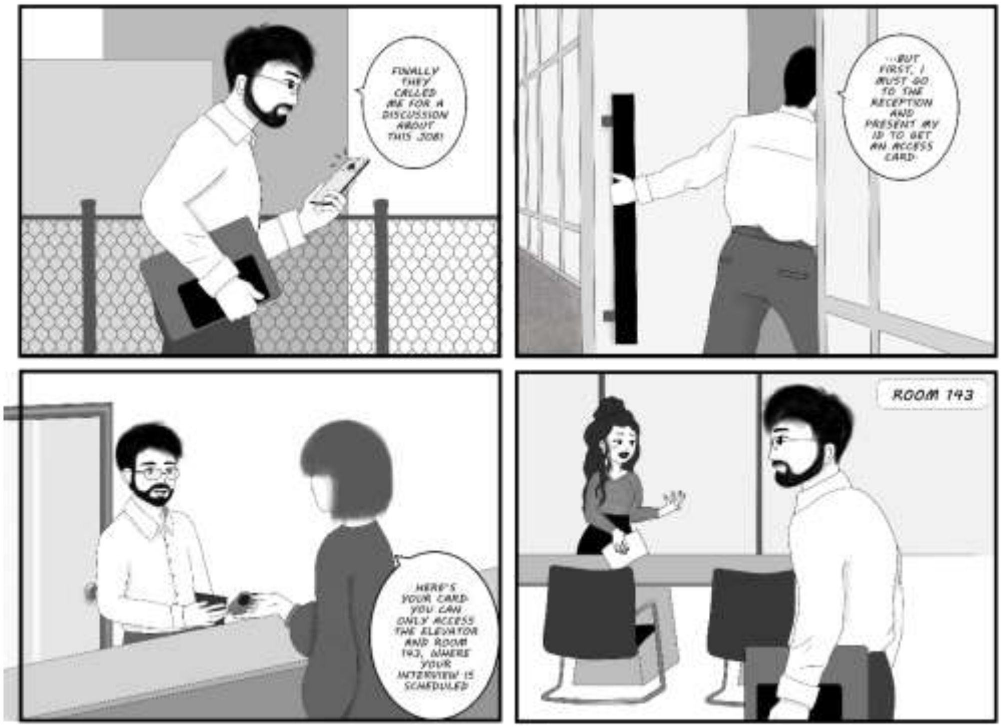

FINALLY THEY CALLED ME FOR A DISCUSSION ABOUT THIS JOB!
...BUT FIRST, I MUST GO TO THE RECEPTION AND PRESENT MY ID TO GET AN ACCESS CARD
HERE'S YOUR CARD YOU CAN ONLY ACCESS THE ELEVATOR AND ROOM 143, WHERE YOUR INTERVIEW IS SCHEDULED
ROOM 143

Figure 1.1 The OAuth 2 specification is very similar to accessing an office building.

The process of getting into the building for the discussion is very similar to how authentication and authorization work in an OAuth 2 implementation. You are the user who needs to execute a specific use case (go to a particular room for the discussion). For that, you use your credentials (the ID) to authenticate at the front desk (the authorization server). Once you prove who you are, you have an access card (the token). But you can use this token only to access specific resources (like elevators and specific doors). You can use the access card only for a short period. After your discussion, you must return the card to the front desk.

In this section, we discuss the responsibilities that interact with one another in an OAuth 2 system, and you'll find out how is it similar to visiting an organization's headquarters for an interview. We also discuss what OAuth 2 is as a specification and the difference between OpenID Connect (a protocol) and OAuth 2 (the specification it relies on)..

First, let's discover who plays roles in an OAuth 2 system. Figure 1.2 presents the main actors of an OAuth 2 system. By actor, I mean any entity that plays a role in the system's functionality. With an OAuth 2 system, you'll find the following actors:

- **The user** – the person that uses the application. The users usually work with a frontend application which we call a client. The users don't always exist in an OAuth 2 system, as we discuss in section 1.3.3 where you'll learn about the client credentials grant type.

- **The client** – the application that calls a backend and needs authentication and authorization. The client can be a web app, a mobile app, or even a desktop app or a separate backend service. When the client is a backend service, the system usually doesn't have a user actor.
- **The resource server** – A backend application that authorizes and serves calls sent by one or more client applications.
- **The authorization server** – An app that implements authentication and safe storage of credentials.

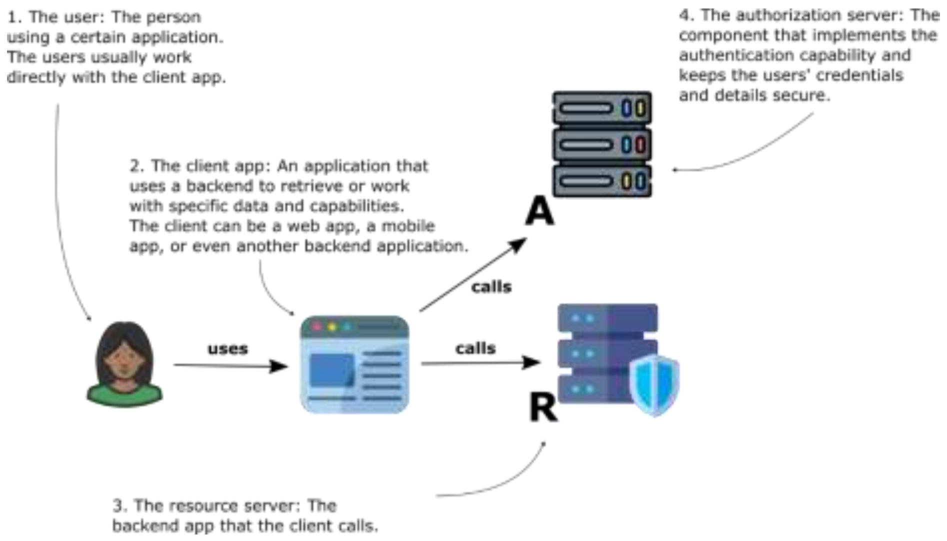

1. The user: The person using a certain application. The users usually work directly with the client app.
4. The authorization server: The component that implements the authentication capability and keeps the users' credentials and details secure.
2. The client app: An application that uses a backend to retrieve or work with specific data and capabilities. The client can be a web app, a mobile app, or even another backend application.
A
calls
uses
calls
R
3. The resource server: The backend app that the client calls.

Figure 1.2 The actors of an OAuth 2 system. The users use a client that needs to get authorized for specific actions on the backend service (which we call a resource server). To get authorized by its backend, the client requests first need to be authenticated by the authorization server.

Let's now discuss about how authentication and authorization really happen. In just a few words, the steps are simple.

1. The user executes a certain use case with the client app.
2. The client app gets authorized to call the resource server to serve the user's request.
3. To get authorized, the client first requests a token (called an access token) from the authorization server. This token is just some specific information that helps the client to prove the authorization server correctly identified them.
4. The client uses the token the authorization server issues to get authorized when sending requests to its backend (the resource server).

Figure 1.3 details a bit of the flow in a visual way. The numbered steps in the figure represent the following:

1. The user tries to use the client application to execute a particular use case.
2. The client application knows it can't call its backend without first having a token that will allow it to get authorized. The client requests such an access token from the authorization server.
3. Following the client app's request, the authorization server issues a token and sends it to the client app.
4. The client uses the app to send requests to its backend (the resource server).
5. The resource server authorizes the client's request. If authorized successfully, the resource server executes the client's request and replies back.
6. The client shows the result to the user.

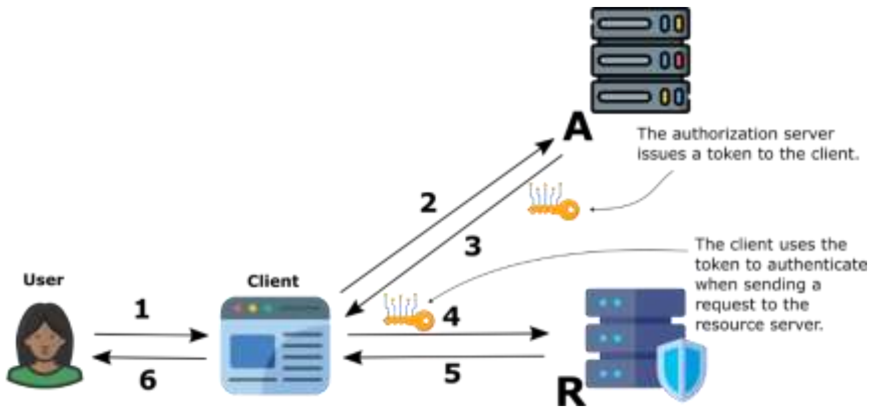

A
The authorization server issues a token to the client.
2
3
The client uses the token to authenticate when sending a request to the resource server.
User
Client
1
6
4
5
R

Figure 1.3 The most straightforward way to describe the OAuth 2 authentication process. The client requests a token from the authorization server and uses this token to get authorized when sending requests to the backend app (resource server).

But what exactly is this token that the authorization server issues? A token can be any piece of data (usually a string of characters) that allows the client to prove they (and/or the user) have been identified by the authorization server. The token is also a way to get more details about both user and client if needed. Since the authorization server now manages all the user and client details, the backend sometimes needs to get part of these details from the authorization server and use them. The backend will get such details through the means of the token. Sometimes the token itself contains the needed details (as you'll read in section 1.2, such tokens are called non-opaque tokens), otherwise the backend needs to call the authorization server to get the data about the client and the user (opaque tokens). Also, unlike a physical key, an access token doesn't have a large lifespan. An access token expires after a short period (in most cases minutes), after which the client needs to ask the authorization server again for another token. This way, a lost token (such as a lost key) can't be misused.

OAuth 2 describes multiple flows in which a client might get a token. We call these flows "grant types" and we discuss the most common grant types used in section 1.3.

1.2 Using various token implementations

Tokens are the access cards (figure 1.4) a client uses to get authorized when sending requests to the backend (the resource server). Tokens are an essential part of the OAuth 2 authentication and authorization process since they're the ones used to prove the authenticity of a client and user authentication, but they are also the way a backend gets more details about the client and the user.

In this section, we discuss how tokens are classified and, depending on the token type, how they are used in the authorization process.

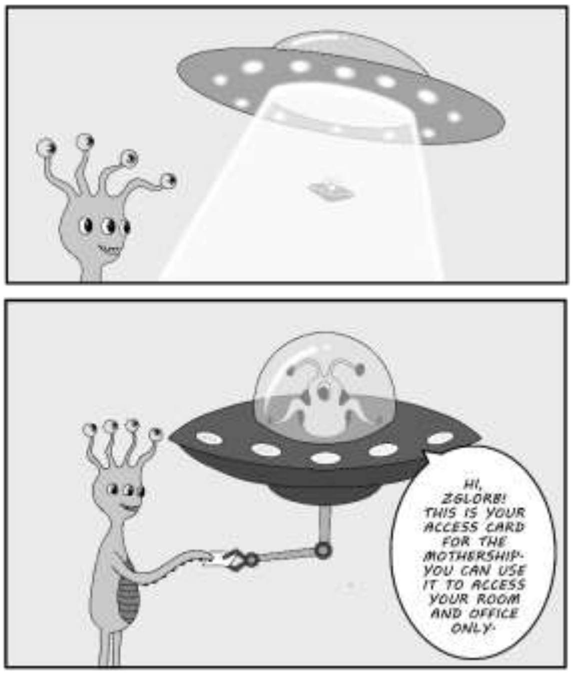

HI, ZGLORB! THIS IS YOUR ACCESS CARD FOR THE MOTHERSHIP. YOU CAN USE IT TO ACCESS YOUR ROOM AND OFFICE ONLY.

Figure 1.4 Zglorb (the user) needs to access the Mothership (resource server). To do that, they're first authenticated (by the authorization server) and then they're given an access card (the token). Zglorb can only access specific areas (resources) of the Mothership using their access card.

We classify tokens based on the way they provide the resource server with the data for authorization:

- Opaque – tokens which don't store data. To implement the authorization, the resource server usually needs to call the authorization server, provide the opaque token, and get the details. This call is known as the introspection call.
- Non-opaque – tokens that store data making it immediately available to the backend to implement the authorization. The most known non-opaque token implementation is the JSON Web Token (JWT).

1.2.1 Using opaque tokens

Opaque tokens don't contain data that the backend can use to identify either the user or the client, or to implement the authorization rules. Opaque tokens are just proof of an authentication attempt. When a resource server receives an opaque token, it needs to call the authorization server to find out whether the token is valid and to get more information to allow it to apply authorization constraints.

An opaque token is literally like a key to a treasure chest. It doesn't provide any information up-front; you know it works only when you try opening the door with it. Once you find out it's valid, it also gets you to what's inside the chest (in this case, the user and client details). Figure 1.5 visually describes this analogy.

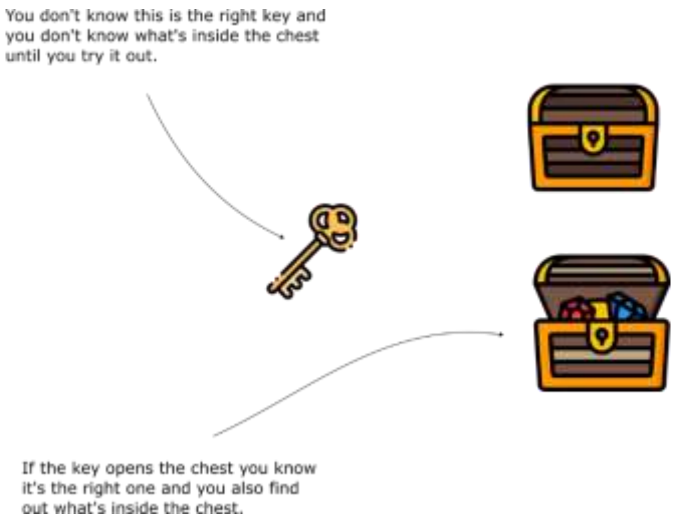

You don't know this is the right key and you don't know what's inside the chest until you try it out.
If the key opens the chest you know it's the right one and you also find out what's inside the chest.

Figure 1.5 An analogy to opaque tokens. An opaque token is just like a key. You don't know if it works until you try it. If the key works, you also get access to what's inside it opens.

The resource server calls an endpoint provided by the authorization server to find out if the opaque token is valid and get the needed details about the client and the user to whom the token was issued. This call is named an introspection call (figure 1.6). Once the resource server has these details, it can apply the authorization constraints.

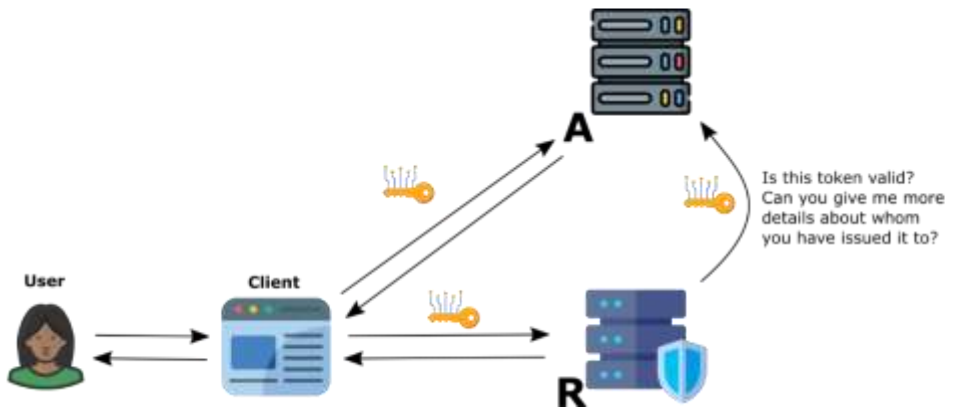


Figure 1.6 The introspection call. The resource server sends a request to the authorization server to find out if the opaque token is valid and details about to whom it was issued.

1.2.2 Using non-opaque tokens

Unlike the opaque tokens we discussed in section 1.2.1, non-opaque tokens contain information about the client and the user to whom the authorization server issued the tokens during the authentication process. You can compare non-opaque tokens with signed documents (figure 1.7).

The details about the user and the client
appear on the document.
One can verify the document is authentic
by checking the signature.

Figure 1.7 The non-opaque token is like a signed document. It contains the needed details for the resource server to apply the authorization constraints and a signature to validate its authenticity.

The most common implementation of a non-opaque token is the JSON Web Token (JWT). A JWT is composed of three parts (figure 1.8):

- The header – Usually contains data about the token, such as the cryptographic algorithm used to sign the token, or the key ID that the authorization server used to sign it.

- The body – Usually contains data about the entity to whom the token was issued, such as the client and the user details.
- The signature – A cryptographically generated value that can be used to prove that the authorization server indeed issued the token and that no one altered its content (in the header or body) after it was generated.

The data in the header and body is JavaScript Object Notation (JSON) formatted, then encoded as Base64 to make it smaller and easier to transfer. The three parts are separated by dots.

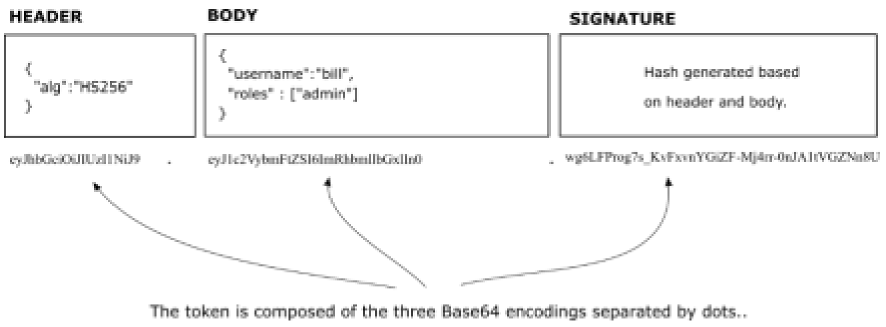

HEADER
BODY
SIGNATURE
{ "alg":"HS256" }
{ "username":"bill", "roles" : ["admin"] }
Hash generated based on header and body.
The token is composed of the three Base64 encodings separated by dots..

Figure 1.8 The anatomy of a JWT token. The header and the body contain the details needed for the resource server to validate the token's authenticity and apply the authorization constraints.

The next snippet shows an example of a JWT where the three parts are Base64 encoded separated by dots.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ik
pvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQ
ssw5c
```

You may be asking yourself now, "When should I use opaque tokens, and when should I use non-opaque ones?" As I told you a bit earlier in this section, non-opaque are the most frequently used today because they don't require introspection to validate them. However, non-opaque tokens contain data, and the client sends this data over the wire to its backend. Anyone obtaining the token can also see the data the token carries. In the large majority of cases, this is not an issue. And I recommend anyone avoid sending too much data within the token.

But what should you do if you have a larger amount of data, or data that is unsafe to send over the wire carried within a token? In such a case, opaque tokens might be a good alternative. I recommend you first consider non-opaque tokens and fall back to opaque ones only if the amount of data to be carried by the token is too large or you want to send details that are more sensitive, and you want to avoid exchanging them through the token.

1.3 Obtaining tokens through various grant types

In this section, we discuss grant types. A grant type is a process in which a client gets a token. In apps, you'll find various approaches in which a client gets a token from the authorization server. We'll discuss the three most used grant types. At the end of this section, we'll explore how a client can re-generate a token after it expires.

> **NOTE** You might still find apps implementing two other grant types: the implicit grant type and the password grant type. These two grant types became deprecated because they were found not to be secure enough. We won't discuss these two grant types in this book; I don't recommend using them in apps. You can always replace either with one of the other grant types discussed in this section. If you're curious to learn more about the password grant type, you find a good discussion in chapter 12 of the first edition of Spring Security in Action. We also go fast through what the implicit grant type was and why it's deprecated when discussion the authorization code grant type in this section.

In section 1.3.1, we discuss the authorization code grant type, which is the most used grant type when the system needs to allow a user to authenticate. In section 1.3.2, we discuss an addition to the authorization code grant type – the proof key for code exchange (PKCE). In section 1.3.3, we continue with the situation where an app needs to get a token without having a user authenticate, and we end with how to re-generate tokens in section 1.3.4.

1.3.1 Getting a token using the authorization code grant type

The authorization code grant type is the most used grant type today. We use this grant type when our app needs to authenticate a user. To easily understand this grant type, see figure 1.9, which visually presents the steps in a sequence diagram.

1. The user wants to do something within the app they use. For example, let's say that the girl on the left side of the diagram is Mary, an accountant who wants to see all the invoices the company she works for needs to pay.
2. The app Mary uses is the client. In this case, Mary stays in front of her computer, so the client app she uses is a web app. But Mary could have also used the mobile version of the app. In both cases, the grant type would have looked the same. Because Mary didn't log in, the app redirects her to a login page hosted by the authorization server.
3. Now, Mary sees the login page in her browser. The login page is not in the app she accessed, but is hosted by a different system. Mary recognizes the page she was redirected to as the central authentication application that she uses for any of the apps she works with for the company. Mary knows that after using her credentials, the browser will return her to the invoices application, and she will be able to see the invoices and operate with the data she needs. Mary fills in the correct credentials and selects the login button.

4. Because the credentials Mary provided are correct, the authorization server redirects back to the initial application. The authorization server also provides the initial application (the client) a unique code named an "authorization code." The client will use this code to get an access token.
5. The client asks for an access token. The client needs this access token to send requests to its backend (the resource server).
6. Because the authorization code is correct (the same one the server provided in step 5), the authorization server responds with an access token.
7. The client app uses the access token to send a request to its backend and get authorized.

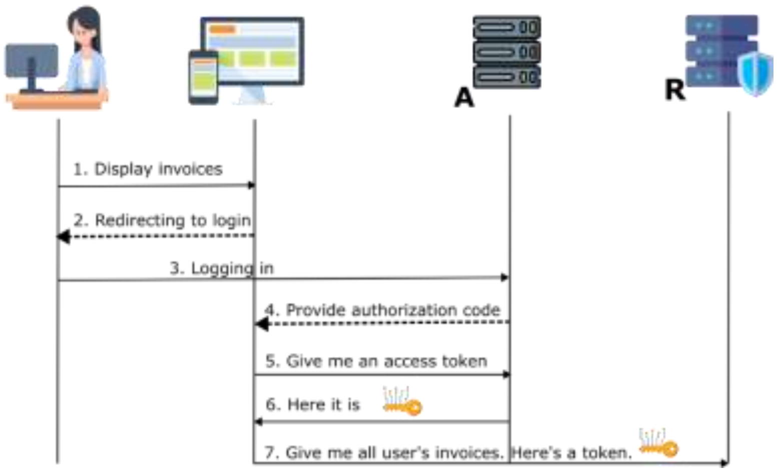

A
R
1. Display invoices
2. Redirecting to login
3. Logging in
4. Provide authorization code
5. Give me an access token
6. Here it is
7. Give me all user's invoices. Here's a token.

Figure 1.9 The authorization code grant type. The user needs to log in. The authorization server provides the client with an authorization code, and the client can use this authorization code to get an access token. The client can use the access token to get its request authorized by the resource server.

A few observations to help you better understand this flow:

- Pay attention to the dotted arrows. It's essential to remember that those represent redirects in browser and not requests or responses. At step 2, the client app redirects the user to the authorization server login page (it redirects in the browser to a web page of another app). At step 4, the authorization server redirects back to the client app providing the authorization code (usually as a query parameter).
- Mary (the user) isn't aware of steps 4 through 7. After she logs in, she will in the end see the invoices displayed by the client who will get them in response to step 7 after being authorized.
- Remember not to confuse the authorization code and the access token. The access token is what the client needs, in the end, to get authorized by its backend (step 7). But to

get the access token, the client first gets an authorization code that they will use to get the access token (step 4).

In addition to the discussed, many developers new to authorization and authentication are confused about step 4. The question I frequently get is, "Why doesn't the authorization server directly return the access token here?" It looks strange that the client needs to take one more step to get the access token when they could have directly gotten it at step 4.

But it makes sense. In fact, in the first version of OAuth, the authorization server provided the access token instead of the authorization code at step 4. This is what we now name the "implicit grant type," which is deprecated and no longer recommended to be used. The reason is a redirect can be easily intercepted, and the access token could have been very easily obtained by a bad-intentioned individual. By returning the authorization code, the authorization server forces the client to re-send a request where they must authenticate again with their credentials. This way, if one intercepts the redirect and gets the authorization code, it's not enough to get an access token. They would need to also know the client credentials to send the request and get the token.

Figure 1.10 visually presents the two steps where the authorization code adds supplementary protection to avoid allowing someone to get the access token.

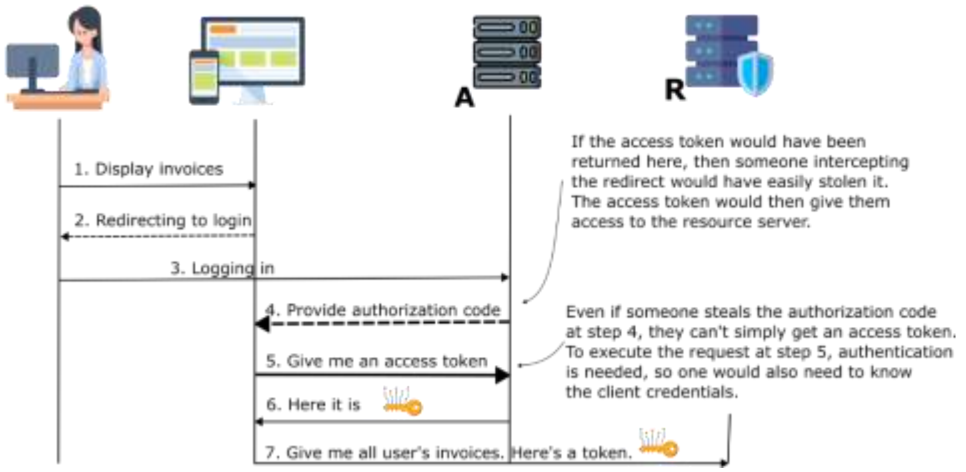

A
R
1. Display invoices
2. Redirecting to login
3. Logging in
4. Provide authorization code
5. Give me an access token
6. Here it is
7. Give me all user's invoices. Here's a token.
If the access token would have been returned here, then someone intercepting the redirect would have easily stolen it. The access token would then give them access to the resource server.
Even if someone steals the authorization code at step 4, they can't simply get an access token. To execute the request at step 5, authentication is needed, so one would also need to know the client credentials.

Figure 1.10 Obtaining an authorization code after login forces the client to execute one more request to get the access token. When sending this request, the client needs to authenticate using their credentials. This approach makes the life more difficult to one who'd want to steal the access token, since they'd need to not only steal the authorization code but also know the client credentials.

1.3.2 Applying PKCE protection to the authorization code grant type

What if a bad-intentioned individual manages to also get the client credentials? In this case, they could obtain an access token and send requests to the resource server. Is there a way in which we can avoid something like this from happening? Yes, the proof key for code exchange

(PKCE, usually pronounced “pixy”) is an enhancement added to the authorization code flow to make it more secure. In this section, we discuss how PKCE covers the case where someone could get an access token by stealing the client credentials.

Using PKCE only affects two of the steps of the authorization code grant type, which we discussed in section 1.3.1. In figure 1.11, I made the arrows representing steps 3 and 5 thicker. These two are the steps of the authorization code grant type where PKCE is applied.

1. First, the client needs to generate a random value. This value can be a random string of bytes. This value is called **the verifier**.
2. Secondly, the client will apply a hash function over the randomly generated value at step 1. A hash function is encryption characterized by the fact that the output cannot be turned back to the input . The result of applying the hash function over the verifier is called **the challenge**.

```
verifier = random();
challenge = hash(verifier);
```

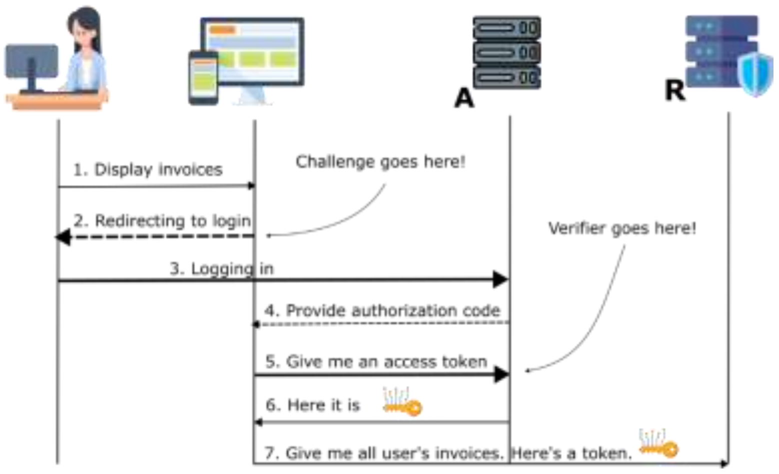


Figure 1.11 The client will send a challenge on step 3 and the verifier on step 5 to prove that they are the same client who initially asked the user to login.

The client sends the challenge in step 3 with the user login. The authorization server keeps the challenge and expects the verifier with the request made at step 5 to get the access token. If the verifier the client sends when requesting the token at step 5 matches with the challenge they sent at step 3, then the authorization server knows the client app that requests the token is the same one that requests the user to authenticate.

One cannot get an access token now even if they somehow manage to get the authorization code at step 4. This is because they would need to know the verifier value as well. They can’t know the verifier since the client has not yet sent it on the wire. And they can’t get the verifier simply by intercepting the challenge (at step 3) because the challenge is created with a hash function, implying that the output can’t be turned back into the input.

1.3.3 Getting a token with the client credentials grant type

Sometimes an app needs to get authorization without user intervention. If there’s no user on the scene, an app will have to use the client grant type to get an access token. This situation commonly happens when a service needs to call another service when an objective event, such as the timer of a scheduled process, triggers it. Using the client grant type, the app only needs to authenticate with its client credentials. Figure 1.12 presents the client credentials grant type.

1. The app requests an access token from the authorization server. The app uses their credentials to authenticate.
2. If credentials are valid, the authorization server issues an access token.
3. The app uses the access token to get authorized when sending a request to the resource server.

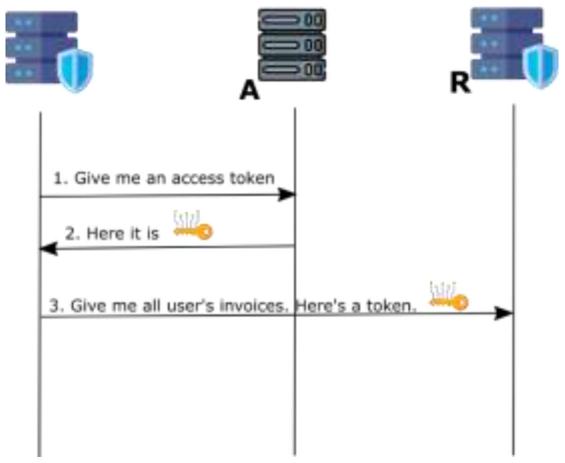

A
R
1. Give me an access token
2. Here it is
3. Give me all user's invoices. Here's a token.

Figure 1.12 The client grant type. An app gets an access token without needing a user to authenticate.

1.3.4 Using refresh tokens to get new access tokens

One essential thing you must remember about tokens is that they must have a relatively short life span. The exact time they stay alive is usually decided according to the scenario, but it’s usually as short as 15 minutes, and I have never used tokens that live more than one hour. Eventually, all tokens need to expire sooner or later. When a token expires, the resource server won’t accept it anymore. In such a case, when a client has a token, and the token has expired, they have two options:

1. Get a new access token by repeating the grant type steps. This implies asking the user to log in again in case of the authorization code grant type.
2. Use a refresh token to get a new access token.

Refresh tokens are particularly useful when the client uses a grant type, such as the authorization code, which implies a user needs to log in. Imagine you have tokens with a 15-minute lifespan. As a user, would you not be bothered if your app asked you to log in again and again every 15 minutes? I would!

The app can use the refresh token to get a new access token instead of asking the user to log in every time the access token expires.

Figure 1.13 shows the steps for using the refresh token:

1. The user tries to get some data, which implies that the client must call its backend.
2. Because the access token (previously obtained) expired, the client needs to get a new access token. The client sends a refresh token to prove they are the same who had already authenticated before.
3. The authorization server recognizes the refresh token and provides the client with a new access token.
4. The client can call the backend (the resource server) and get authorized with the new access token.

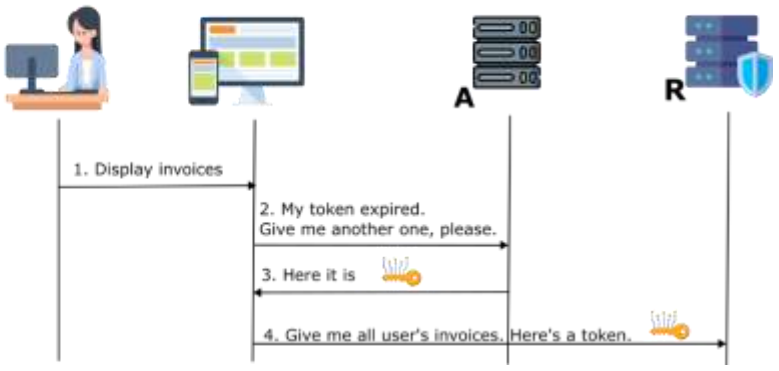

A
R
1. Display invoices
2. My token expired.
Give me another one, please.
3. Here it is
4. Give me all user's invoices. Here's a token.

Figure 1.13 A client app can use a refresh token to get a new access token when the old one expires. This way, the app avoids requesting the user to authenticate again.

1.4 What does OpenID Connect bring to OAuth 2

There's certainly still a lot of confusion out there about OpenID Connect (sometimes named OIDC) and OAuth 2, and the differences between the two. I usually tell my students to stress less about this subject: "If you understand OAuth 2, you also know how to use OpenID Connect."

In fact, OIDC is a protocol built over the OAuth 2 specification. For this reason, understanding OAuth 2 helps you easily get OIDC. Let me give you an analogy for specifications versus protocols.

We all use electrical outlets every day. Electrical outlets look different around the world. Sometimes this is a real pain when you travel. You sometimes need to have adapters to ensure you can charge your devices, especially if you travel between different geographical regions.

But behind the scenes, all outlets work the same way. There're some wires which output the electrical voltage. You can define a framework on which all the electrical outlets in the world work in only a couple of bullet points:

- An electrical outlet has three wires which allow electrical current to flow: the phase, the null, and a grounding. The grounding is optional.
- The electrical outlet provides a voltage that can be either around 120 Volts or 230 Volts.

Now, no worries if you're not a technical folk, you don't need to understand these two bullets. At least not for learning Spring Security. Just take my word for it.

The issue is that even if all sockets in the world fulfill these specifications, we still get situations where we need adapters to travel. The reason is that they don't have a common protocol. Adapters are needed to adapt the outlet from one protocol to another (for example, North America to Europe).

The same happens with apps and authentication and authorization. If two apps fulfill the OAuth 2 specification, they still can run in situations where they are not fully compatible and need to be adapted, because they don't run the same protocol. OpenID Connect is a protocol that restricts a bit the liberty of the OAuth 2 specification introducing a few changes. The major changes are

- Specific values for the scopes (such as "profile" or "openid").
- The use of an extra token named ID token, used to store the details about the identity of the user and client to whom the token was issued.
- Usually the term "grant type" is also referred to as "flow" when discussing it in terms of OIDC, while the "authorization server" is commonly called "Identity provider" or "IdP."

1.5 The sins of OAuth 2

In this section, we discuss possible vulnerabilities of the applications using OAuth 2 authentication and authorization. It's essential to understand what can go wrong when using OAuth 2 to avoid these scenarios when developing your applications. Of course, like anything else in software development, OAuth 2 isn't bulletproof. It has its vulnerabilities, which we must be aware of when building our applications. I enumerate here some of the most common:

- Using Cross-Site Request Forgery (CSRF) on the Client – With a user logged in, CSRF is possible if the application doesn't apply any CSRF protection mechanism.
- Stealing client credentials – Storing or transferring the credentials unprotected can create breaches that allow attackers to steal and use them.

- Replaying tokens – As discussed in section 1.2, tokens are the "keys" we use within an OAuth 2 authentication and authorization architecture to access the resources. You send them over network, and sometimes they might be intercepted. If intercepted, they are stolen and can be reused. Imagine you lose the key from your home's front door. What could happen? Somebody else could use it to open the door as many times as they like (replay).
- Token hijacking - You interfere in the authentication process and steal tokens you can use to access resources. This is also a potential vulnerability of using refresh tokens, as they, as well, can be intercepted and used to obtain new access tokens. I recommend this helpful article http://blog.intothesymmetry.com/2015/06/on-oauth-token-hijacks-for-fun-and.html.

Remember, OAuth 2 is a framework. The vulnerabilities are the result of wrongly implementing functionality over it. Using Spring Security already helps us mitigate most of those vulnerabilities in our applications.

For more details on vulnerabilities related to the OAuth 2 framework and how a bad-intentioned individual could exploit them, you find a great discussion in part 3 of *OAuth 2 In Action* by Justin Richer and Antonio Sanso (Manning, 2017): https://livebook.manning.com/book/oauth-2-in-action/part-3.

1.6 Summary

- The OAuth 2 framework describes secure ways in which a backend can authenticate its clients. OpenID Connect is a protocol that implements the OAuth 2 client by applying some constraints for the possible implementations.
- The four main actors in an OAuth 2 system are
  - The user – A person who wants to execute a use case.
  - The client – An app that must be authorized to access a resource or use case on a given backend.
  - The resource server – A backend that needs to authorize a client to execute a specific use case or access a resource.
  - The authorization server – An app that manages the user and client details allowing them to authenticate and providing a token that can be used for means of authorization.
- The token is an access card (or a key) a client gets from the authorization server and uses to get authorized to call a use case or access a specific resource on a secured backend (the resource server).
- We classify tokens into two categories:
  - Opaque – tokens that don't contain details about the user and client for whom they were issued. For such tokens, a resource server always needs to call the authorization server to validate the token and get the details it needs to

    authorize the request. This request for the token validation is called introspection.

    - Non-opaque - tokens that contain details about the user and the client to whom they’ve been issued. The most common implementation of non-opaque tokens is the JSON Web Token (JWT).

- There are multiple flows in which a client app can ask the authorization server for a token. These flows in which the token is issued are called grant types. The most common grant types are
    - The authorization code grant type
    - The client credentials grant type
- Sometimes we add supplementary security to the authorization code grant type using a proof key for code exchange (PKCE) approach. In this approach, the client uses additional values to avoid someone being able to obtain access tokens by stealing the client credentials and the authorization code.
- In specific cases, an app might need to get new access tokens without having the user re-authenticate. For such cases, the app can use refresh tokens. Refresh tokens are special tokens that can only be used to get new access tokens.

Implementing an OAuth 2 authorization server

This chapter covers

- Implementing a Spring Security OAuth 2 authorization server
- Using the authorization code and client credentials grant types
- Configuring opaque and non-opaque access tokens
- Using token revocation and introspection

We went through OAuth 2 and OpenID Connect. We discussed the actors that play a role in a system where the authentication and authorization are based on the OAuth 2 specification. The authorization server was one of these actors. Its role is to authenticate a user and the app they use (the client) and issue tokens that the client uses as proof of authentication to access resources protected by a backend. Sometimes the client does that on behalf of a user.

The Spring ecosystem offers a fully customizable way to implement an OAuth 2/OpenID Connect authorization server. The Spring Security authorization server is the de facto way to implement an authorization server using Spring today. In this chapter, we review the main capabilities this framework offers and implement a custom authorization server. Figure 2.1 reminds you about the OAuth 2 actors and the authorization server role we discussed in chapter 1.

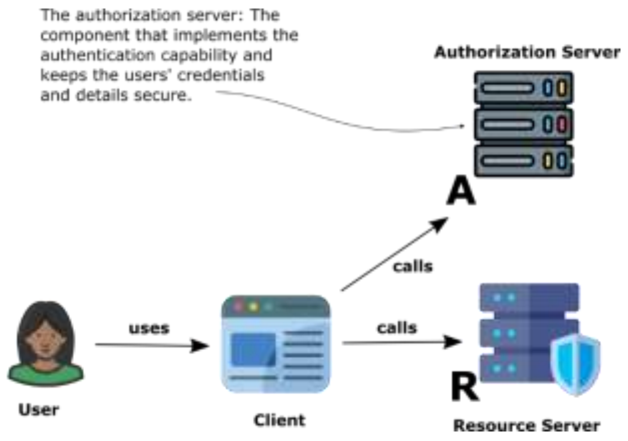

The authorization server: The component that implements the authentication capability and keeps the users' credentials and details secure.
Authorization Server
A
calls
uses
calls
R
User
Client
Resource Server

Figure 2.1 The actors in the OAuth 2 scene. The authorization server protects the users' and clients' details and issues tokens the client can use to get authorized when calling the resource server endpoints.

We start with implementing a simple example in section 2.1. This example uses the default configurations. The default configuration implies that the authorization server will issue non-opaque tokens. In section 2.2, we prove that our implementation works with the authorization code grant type, and then, in section 2.3, we demonstrate the client credentials grant type as well. We continue, in section 2.4, with configuring the authorization server to work with opaque tokens and introspection. We end this chapter's discussion in section 2.5 with token revokation.

Before starting, I'd like also to make you aware that the way to implement an authorization server with Spring Security changed entirely in the past years. In this chapter, we discuss the new approach, but you might also need to know how to implement an authorization server with the older way (for example, if you need to work on an existing app that hasn't been upgraded). In that case, I recommend reading chapter 13 of this book's first edition.

2.1 Implementing a basic authentication using JWTs

In this section, we implement a basic OAuth 2 authorization server using the Spring Security authorization server framework. We go through all the main components you need to plug into the configuration to make this work and discuss them individually. Then, we test the app using the most essential two OAuth 2 grant types: the authorization code grant type and the client credentials grant type.

The main components you need to set up for your authorization server to work properly are

5. *The configuration filter for protocol endpoints* - helps you define configurations specific to the authorization server capabilities, including various customizations (which we'll discuss in section 2.3).

6. *The authentication configuration filter* – similar to any web application secured with Spring Security, you'll use this filter to define the authentication and authorization configurations as well as configurations to any other security mechanisms such as CORS and CSRF.
7. *The user details management components* – same as for any authentication process implemented with Spring Security – are established through a `UserDetailsService` bean and a `PasswordEncoder`.
8. *The client details management* – the authorization server uses a component typed `RegisteredClientRepository` to manage the client credentials and other details.
9. *The key-pairs (used to sign and validate tokens) management* – when using non-opaque tokens, the authorization server uses a private key to sign the tokens. The authorization server also offers access to a public key that the resource server can use to validate the tokens. The authorization server manages the private-public key pairs through a "key source" component.
10. *The general app settings* – a component typed `AuthorizationServerSettings` helps you configure generic customizations such as the endpoints the app exposes.

Figure 2.2 offers a visual perspective of the components we need to plug in and configure for a minimal authorization server app to work.

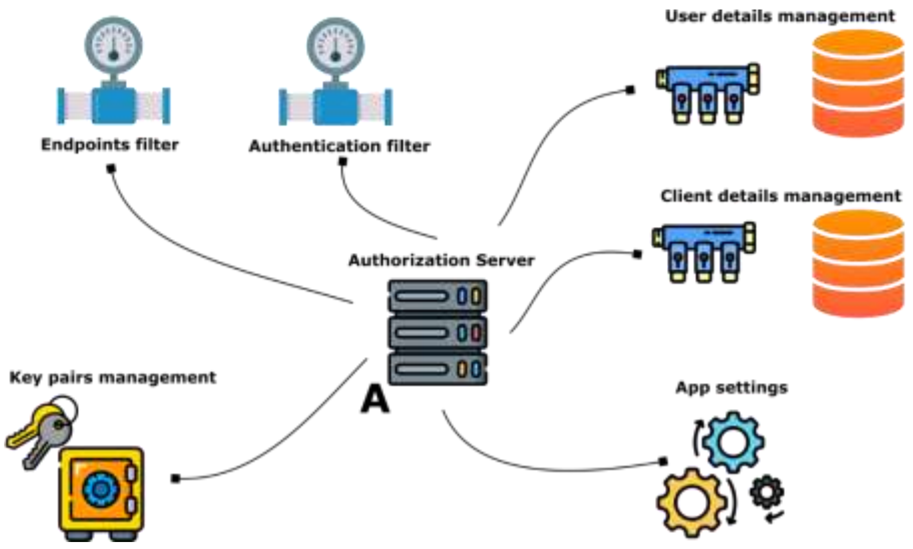


Figure 2.2 The components we need to configure and plug in for an authorization server implemented with Spring Security to work.

To begin, we have to add the needed dependencies to our project. In the next code snippet, you find the dependencies you need to add to your pom.xml project.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-oauth2-authorization-server</artifactId>
</dependency>
```

We write the configurations in standard Spring configuration classes like in the next code snippet.

```
@Configuration
public class SecurityConfig {


}
```

Remember that like for any other Spring app, the beans can be defined in multiple configuration classes, or they can be (depending on the case) defined using stereotype annotations. If you need a refresher on managing the Spring context, I recommend you read the first part of *Spring Start Here* (Manning, 2021), another book I wrote.

Take a look at listing 2.1, which presents the configuration filter for protocol endpoints. The `applyDefaultSecurity()` method is a utility method we use to define a minimal set of configurations you can override later if needed. After calling this method, the listing shows how to enable the OpenID Connect protocol using the `oidc()` method of the `OAuth2AuthorizationServerConfigurer` configurer object.

Also, the filter in listing 2.1 specifies the authentication page to which the app needs to redirect the user when asked to log in. We need this configuration since we expect to enable the authorization code grant type for our example, which implies that the users must authenticate. The default path in a Spring web app is /login, so unless we configure a custom one, this is the one we'll use for the authorization server configuration.

Listing 2.1 Implementing the filter for configuring the protocol endpoints

```
@Bean
@Order(1)
public SecurityFilterChain asFilterChain(HttpSecurity http)
  throws Exception {

    OAuth2AuthorizationServerConfiguration     #A
       .applyDefaultSecurity(http);     #A

    http.getConfigurer(OAuth2AuthorizationServerConfigurer.class)     #B
        .oidc(Customizer.withDefaults());    #B

    http.exceptionHandling((e) ->
      e.authenticationEntryPoint(
         new LoginUrlAuthenticationEntryPoint("/login"))      #C
    );

    return http.build();
  }
```

#A Calling the utility method to apply default configurations for the authorization server endpoints.

#B Enabling the OpenID Connect protocol.
#C Specifying the authentication page for users.

Listing 2.2 configures authentication and authorization. These configurations work similarly to any web app In listing 2.2, I set up the minimum configurations:

1. Enabling the form login authentication so the app gives the user a simple login page to authenticate.
2. Specify that the app only allows access to authenticated users to any endpoints.

Other configurations you could write here, besides authentication and authorization, could be for specific protection mechanisms such as CSRF or CORS.

Observe also the `@Order` annotation I used in both listings 2.1 and 2.2. This annotation is needed since we have multiple `SecurityFilterChain` instances configured in the app context, and we need to provide the order in which they take priority in configuration.

Listing 2.2 Implementing the filter for authorization configuration

```
@Bean
@Order(2)      #A
public SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http)
  throws Exception {

  http.formLogin(Customizer.withDefaults());    #B

  http.authorizeHttpRequests(    #C
    c -> c.anyRequest().authenticated()     #C
  );       #C


  return http.build();
}
```

#A We set the filter to be interpreted after the protocol endpoints one.
#B We enable the form login authentication method.
#C We configure all endpoints to require authentication.

If you expect clients will use the authorization server you build for grant types that imply user authentication (such as the authorization code grant type), then your server needs to manage user details! All you need is a `UserDetailsService` and a `PasswordEncoder` implementation.

Listing 2.3 presents a definition for these two components. In this example, we use an in-memory implementation for the `UserDetailsService`, but remember you learned how to write a custom implementation for it in chapter 3. In most cases, same as for other web apps, you'll keep such details stored in a database. Therefore, you'll need to write a custom implementation for the `UserDetailsService` contract.

The `NoOpPasswordEncoder` doesn't transform the passwords anyhow, leaving them in clear text and at the disposal of anyone who can access them, which is not good. You should always use a password encoder with a strong hash function such as BCrypt.

Listing 2.3 Defining the user details management

```
@Bean
public UserDetailsService userDetailsService() {
  UserDetails userDetails = User.withUsername("bill")
         .password("password")
         .roles("USER")
         .build();

  return new InMemoryUserDetailsManager(userDetails);
}

@Bean
public PasswordEncoder passwordEncoder() {
  return NoOpPasswordEncoder.getInstance();
}
```

The authorization server needs a `RegisteredClientRepository` component to manage the clients' details. The `RegisteredClientRepository` interface works similarly to the `UserDetailsService` one, but it's designed to retrieve client details. Similarly, the framework provides the `RegisteredClient` object, whose purpose is to describe a client app the authorization server knows.

We can say that `RegisteredClient` is for clients what `UserDetails` is for users. Similarly, `RegisteredClientRepository` works for client details like a `UserDetailsService` works for the users' details (figure 2.3).

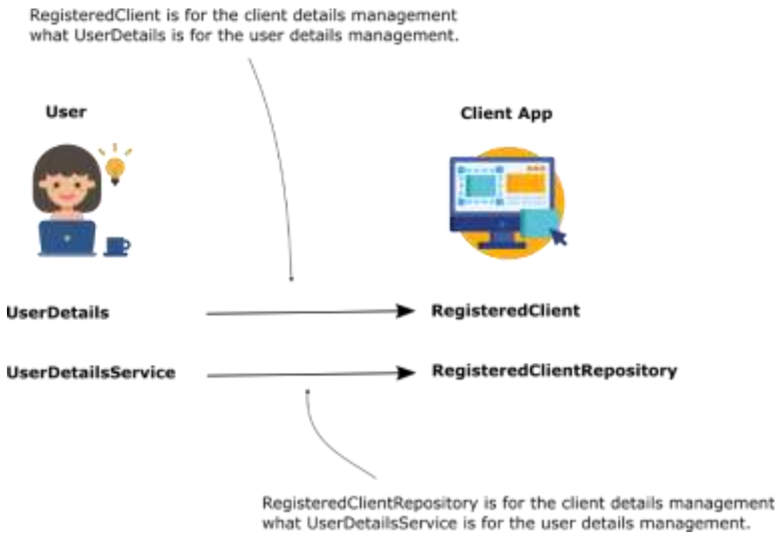

RegisteredClient is for the client details management what UserDetails is for the user details management.
User
Client App
UserDetails
RegisteredClient
UserDetailsService
RegisteredClientRepository
RegisteredClientRepository is for the client details management what UserDetailsService is for the user details management.

Figure 2.3 To manage the clients' details, we use a RegisteredClientRepository implementation. The RegisteredClientRepository uses RegisterdClient objects to represent the clients' details.

In this example, we'll use an in-memory implementation to allow you to focus on the overall implementation of the authorization server. Still, in a real-world app, you'd most likely need to provide an implementation to this interface to grab the data from a database. For this to work, you implement the `RegisteredClientRepository` interface similarly to how you learned to implement the `UserDetailsService` interface.

Listing 2.4 shows the definition of the in-memory `RegisteredClientRepository` bean. The method creates one `RegisteredClient` instance with the needed details and stores it in-memory to be used during authentication by the authorization server.

Listing 2.4 Implementing client details management

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
  RegisteredClient registeredClient =        #A
    RegisteredClient      #A
        .withId(UUID.randomUUID().toString())        #A
        .clientId("client")       #A
        .clientSecret("secret")        #A
        .clientAuthenticationMethod(
           ClientAuthenticationMethod.CLIENT_SECRET_BASIC)         #A
        .authorizationGrantType(       #A
           AuthorizationGrantType.AUTHORIZATION_CODE)         #A
        .redirectUri("https://www.manning.com/authorized")        #A
        .scope(OidcScopes.OPENID)        #A
        .build();       #A

  return new InMemoryRegisteredClientRepository(registeredClient);      #B
}
```

#A Creating a RegisteredClient instance.
#B Adding it to be managed by the in-memory RegisteredClientRepository implementation.

The details we specified when creating the `RegisteredClient` instance are the following:

- *A unique internal ID* – value that uniquily identifies the client and has a purpose only in the internal app processes.
- *A client ID* - external client identifier similar to what a username is for the user.
- *A client secret* – similar to what a password is for a user.
- *The client authentication method* – tells how the authorization server expects the client to authenticate when sending requests for access tokens.
- *Authorization grant type* – A grant type allowed by the authorization server for this client. A client might use multiple grant types.
- *Redirect URI* – One of the URI addresses the authorization server allows the client to request a redirect for providing the authorization code in case of the authorization code grant type.
- *A scope* – Defines a purpose for the request of an access token. The scope can be used later in authorization rules.

In this example, the client only uses the authorization code grant type. But you can have clients that use multiple grant types. In case you want a client to be able to use multiple grant types, you need to specify them as presented in the next code snippet. The client defined in the next code snippet can use any of the authorization code, client credentials, or the refresh token grant types.

```
RegisteredClient registeredClient =
    RegisteredClient
        .withId(UUID.randomUUID().toString())
        .clientId("client")
        .clientSecret("secret")
        .clientAuthenticationMethod(
           ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
        .authorizationGrantType(
           AuthorizationGrantType.AUTHORIZATION_CODE)
        .authorizationGrantType(
           AuthorizationGrantType.CLIENT_CREDENTIALS)
        .authorizationGrantType(
           AuthorizationGrantType.REFRESH_TOKEN)
        .redirectUri("https://www.manning.com/authorized")
        .scope(OidcScopes.OPENID)
        .build();
```

Similarly, by repeatedly calling the `redirectUri()` method, you can specify multiple allowed redirect URIs. In a similar way, a client might also have access to multiple scopes as well.

In a real-world app, the app would keep all these details in a database from where your `RegisteredClientRepository` custom implementation would retrieve them.

Besides having user and client details, you must configure key pair management if the authorization server uses non-opaque tokens. For non-opaque tokens, the authorization server uses private keys to sign the tokens and provides the clients with public keys they can use to validate the tokens' authenticity.

The `JWKSource` is the object providing keys management for the Spring Security authorization server. Listing 2.5 shows how to configure a `JWKSource` in the app's context. For this example, I create a key pair programmatically and add it to the set of keys the authorization server can use. In a real-world app, the app would read the keys from a location where they're safely stored (such as a vault configured in the environment).

Configuring an environment that perfectly resembles a real-world system would be too complex, and I prefer you focus on the authorization server implementation. However, remember that in a real app, it doesn't make sense to generate new keys every time the app restarts (like it happens in our case). If that happens for a real app, every time a new deployment occurs, the tokens that were already issued would not work anymore (since they can't be validated anymore with the existing keys).

So, for our example, generating the keys programmatically works and will help us demonstrate how the authorization server works. In a real-world app, you must keep the keys secured somewhere and read them from the given location.

Listing 2.5 Implementing the key pair set management

```
@Bean
public JWKSource<SecurityContext> jwkSource()
  throws NoSuchAlgorithmException {

  KeyPairGenerator keyPairGenerator =     #A
```

```
        KeyPairGenerator.getInstance("RSA");    #A

        keyPairGenerator.initialize(2048);    #A
        KeyPair keyPair = keyPairGenerator.generateKeyPair();     #A

        RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();     #A
        RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();    #A

        RSAKey rsaKey = new RSAKey.Builder(publicKey)    #A
            .privateKey(privateKey)     #A
            .keyID(UUID.randomUUID().toString())     #A
            .build();     #A

        JWKSet jwkSet = new JWKSet(rsaKey);     #B
        return new ImmutableJWKSet<>(jwkSet);     #C
    }
```

#A Generating a public-private key pair programmatically using the RSA cryptographic algorithm.
#B Adding the key pair to the set the authorization server uses to sign the issued tokens.
#C Wrapping the key set into a JWKSource implementation and returning it to be added to the Spring context.

Finally, the last component we need to add to our minimal configuration is an `AuthorizationServerSettings` object (listing 2.6). This object allows you to customize all the endpoints paths that the authorization server exposes. If you simply create the object as shown in listing 2.6, the endpoints paths will get some defaults that we'll analyze later in this section.

Listing 2.6 Configuring the authorization server generic settings

```
@Bean
public AuthorizationServerSettings authorizationServerSettings() {
  return AuthorizationServerSettings.builder().build();
}
```

Now, we can start the app and test if it works. In section 2.2 we'll run the authorization code flow. Then, in section 2.3 we'll test that the client credentials flow works as expected with our authorization code implementation.

2.2 Running the authorization code grant type

In this section, we test the authorization server we implemented in section 2.1. We expect that by using the registered client details, we'd be able to follow the authorization code flow and get an access token. We'll follow the next steps:

3. Check the endpoints that the authorization server exposes.
4. Use the authorization endpoint to get an authorization code.
5. Use the authorization code to get an access token.

The first step is to find out the endpoints' paths the authorization server exposes. Since we didn't configure custom ones, we must use the defaults. But which are the defaults? You can call the OpenID configuration endpoint in the next snippet to discover these details. This request uses the HTTP GET method, and no authentication is required.

```
http://localhost:8080/.well-known/openid-configuration
```

When calling the OpenID configuration endpoint you should get a response that looks like the one presented in listing 2.7.

Listing 2.7 The response of the OpenID configuration request

```
{
    "issuer": "http://localhost:8080",
    "authorization_endpoint": "http://localhost:8080/oauth2/authorize",
#A
    "token_endpoint": "http://localhost:8080/oauth2/token",     #B
    "token_endpoint_auth_methods_supported": [
        "client_secret_basic",
        "client_secret_post",
        "client_secret_jwt",
        "private_key_jwt"
    ],
    "jwks_uri": "http://localhost:8080/oauth2/jwks",    #C
    "userinfo_endpoint": "http://localhost:8080/userinfo",
    "response_types_supported": [
        "code"
    ],
    "grant_types_supported": [
        "authorization_code",
        "client_credentials",
        "refresh_token"
    ],
    "revocation_endpoint": "http://localhost:8080/oauth2/revoke",
    "revocation_endpoint_auth_methods_supported": [
        "client_secret_basic",
        "client_secret_post",
        "client_secret_jwt",
        "private_key_jwt"
    ],
    "introspection_endpoint": "http://localhost:8080/oauth2/introspect",
#D
    "introspection_endpoint_auth_methods_supported": [
        "client_secret_basic",
        "client_secret_post",
        "client_secret_jwt",
        "private_key_jwt"
    ],
    "subject_types_supported": [
        "public"
    ],
    "id_token_signing_alg_values_supported": [
        "RS256"
    ],
    "scopes_supported": [
        "openid"
    ]
}
```

#A The authorization endpoint to which a client will redirect the user to authenticate.
#B The token endpoint the client will call to request an access token.
#C The key set endpoint a resource server will call to get the public keys it can use to validate tokens.
#D The introspection endpoint a resource server can call to validate opaque tokens.

Look at figure 2.4 to remember the authorization code flow. We'll use this now to demonstrate that the authorization server we built works fine.

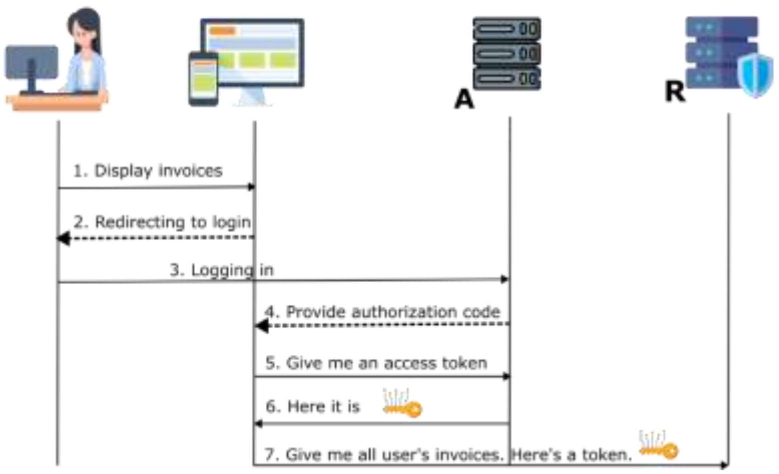

A
R
1. Display invoices
2. Redirecting to login
3. Logging in
4. Provide authorization code
5. Give me an access token
6. Here it is
7. Give me all user's invoices. Here's a token.

Figure 2.4 The authorization code grant type. After the user successfully authenticates, the client gets an authorization code. The client uses the authorization code to get an access token to help it access resources that the resource server protects.

Since we don't have a client for our example, we need to act like one. Now that you know the authorization endpoint, you can put it in a browser address to simulate how the client would redirect the user to it. The next snippet shows you the authorization request.

```
http://localhost:8080/oauth2/authorize?response_type=code&client_id=client&
scope=openid&redirect_uri=https://www.manning.com/authorized&code_challenge
=QYPAZ5NU8yvtlQ9erXrUYR-T5AGCjCF47vN-KsaI2A8&code_challenge_method=S256
```

For the authorization request, you observe I added a few parameters:

- `response_type=code` – this request parameter specifies to the authorization server that the client wants to use the authorization code grant type. Remember that a client might have configured multiple grant types. It needs to tell the authorization server which grant type it wants to use.
- `client_id=client` – the client identifier is like the "username" for the user. It uniquily identifies the client in the system.
- `scope=openid` – Specifies which scope the client wants to be granted with this authentication attempt.
- `redirect_uri=`https://www.manning.com/authorized – Specifies the URI to which the authorization server will redirect after a successful authentication. This URI must be one of those previously configured for the current client.
- `code_challenge=QYPAZ5NU8yvtlQ…` - If using the authorization code enhanced with PKCE, you must provide the code challenge with the authorization request. When

requesting the token, the client must send the verifier pair to prove they are the same application that initially sent this request. The PKCE flow is enabled by default.

- `code_challenge_method=S256` – This request parameter specifies which is the hashing method that has been used to create the challenge from the verifier. In this case, S256 means SHA-256 was used as a hash function.

I recommend you use the authorization code grant type with PKCE, but if you really need to disable the PKCE enhancement of the flow, you can do that as presented in the next code snippet. Observe the `clientSettings()` method that takes a `ClientSettings` instance where you can specify you disable the proof key for code exchange.

```
RegisteredClient registeredClient = RegisteredClient
        .withId(UUID.randomUUID().toString())
        .clientId("client")
        // …
        .clientSettings(ClientSettings.builder()
            .requireProofKey(false)
            .build())
        .build();
```

In this example, we'll demonstrate the authorization code with PCKE, which is the default and recommended way. By sending the authorization request through the browser's address bar, we simulate step 2 from figure 2.4. The authorization server will redirect us to its login page, and we can authenticate using the user's name and password. This is step 3 from figure 2.4. Figure 2.5 shows the login page the authorization server presents to the user.

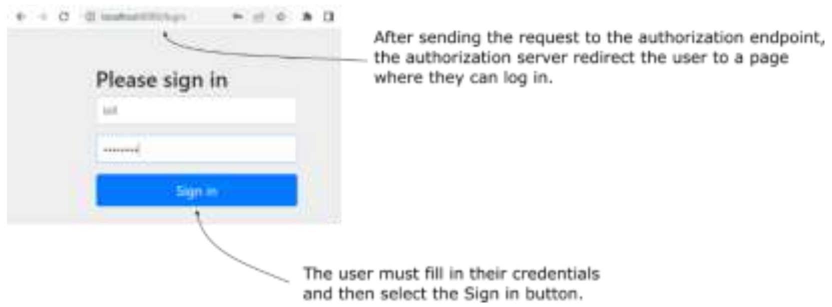

Please sign in
Sign in
After sending the request to the authorization endpoint, the authorization server redirect the user to a page where they can log in.
The user must fill in their credentials and then select the Sign in button.

Figure 2.5 The login page that the authorization server presents to the user in response to the authorization request.

For our implementation, we only have one user (see listing 2.3). Their credentials are username "bill" and the password "password". Once the user fills in the correct credentials and selects the "Sign in" button, the authorization server redirects the user to the requested redirect URI and provides an authorization code (as presented in figure 2.6) – step 4 in figure 2.4.

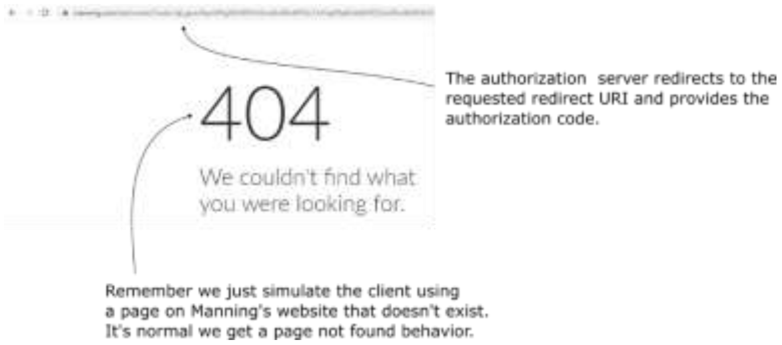

The authorization server redirects to the requested redirect URI and provides the authorization code.
404
We couldn't find what you were looking for.
Remember we just simulate the client using a page on Manning's website that doesn't exist. It's normal we get a page not found behavior.

Figure 2.6 After successful authentication, the authorization server redirects the user to the requested redirect URI and provides an authorization code. The client can use the authorization code to get an access token.

Once the client has the authorization code, it can request an access token. The client can request the access token using the token endpoint. Next snippet shows a cURL request of a token. The request uses the HTTP POST method. Because we specified that HTTP Basic authentication is requested when registering the client, the token request needs authentication with HTTP Basic with the client ID and secret.

```
curl -X POST
'http://localhost:8080/oauth2/token?client_id=client&redirect_uri=https://w
ww.manning.com/authorized&grant_type=authorization_code&code=ao2oz47zdM0D5g
bAqtZVBmdoUWbZI_GXt-
jfGKetWCxaeD1ES9JLx2FtgQZAPMqCZixH4vbhzvuvx2qEJrDnjvOnnojkmYBc8OUMxr0wHkCtE
-NgF1zQ50txKTvPFjTv&code_verifier=qPsH306-
ZDDaOE8DFzVn05TkN3ZZoVmI_6x4LsVglQI' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

The request parameters we used are:

- *client_id=client* – needed to identify the client
- *redirect_uri= https://www.manning.com/authorized* – the redirect URI through which the authorization server provided the authorization code after the successful user authentication.
- *grant_type=authorization_code* – which flow the client uses to request the access token
- *code=ao2oz47zdM0D5…*– the value of the authorization code the authorization server provided to the client
- *code_verifier=qPsH306-ZDD…* – the verifier based on which the challenge that the client sent at authorization was created.

NOTE Pay great attention to all the details. If any of the values doesn't properly match what the app knows or what was sent in the authorization request, the token request won't succeed.

The next snippet shows the response body of the token request. Now the client has an access token it can use to send requests to the resource server.

```
{
    "access_token": "eyJraWQiOiI4ODlhNGFmO…",
    "scope": "openid",
    "id_token": "eyJraWQiOiI4ODlhNGFmOS1…",
    "token_type": "Bearer",
    "expires_in": 299
}
```

Since we enabled the OpenID Connect protocol so we don’t only rely on OAuth 2, an ID token is also present in the token response. If the client had been registered using the refresh token grant type, a refresh token would have also been generated and sent through the response.

Generating the code verifier and challenge

In the example we worked on in this section, I used the authorization code with PKCE. In the authorization and token requests, I used a challenge and a verifier value I had previously generated. I didn’t pay too much attention to these values since they are the client's business and not something the authorization or resource server generates. In a real-world app your JavaScript or mobile app will have to generate both these values when using them in the OAuth 2 flow.

But in case you wonder, I will explain how I generated these two values in this sidebar.

The code verifier is a random 32 bytes piece of data. To make it easy to transfer through a HTTP request, this data needs to be Base64 encoded using an URL encoder and without padding. The next code snippet shows you how to do that in Java.

```
SecureRandom secureRandom = new SecureRandom();
byte [] code = new byte[32];
secureRandom.nextBytes(code);
String codeVerifier = Base64.getUrlEncoder()
        .withoutPadding()
        .encodeToString(code);
```

Once you have the code verifier, you use a hash function to generate the challenge. The next code snippet shows you how to create the challenge using the SHA-256 hash function. Same as for the verifier, you need to use Base64 to change the byte array into a String value making it easier to transfer through the HTTP request.

```
MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");

byte [] digested = messageDigest.digest(verifier.getBytes());
String codeChallenge = Base64.getUrlEncoder()
          .withoutPadding()
          .encodeToString(digested);
```

Now you have a verifier and a challenge. You can use them in the authorization and token requests as discussed in this section.

2.3 Running the client credentials grant type

In this section, we'll try out the client credentials grant type using the authorization server we implemented in section 2.1. Rememeber that the client grant type is a flow that allows the client to get an access token without a user's authentication or consent. Preferably, you shouldn't have a client being able to use both a user-dependent grant type (such as the authorization code) and a client-independent one (such as the client credentials).

As you'll learn in chapter 3 where we discuss the resource server, the authorization implementation might fail to make a difference between an access token obtained through the authorization code grant type and one that the client obtained through the client credential grant type. So it's best to use different registration for such cases and preferably distinguish the token usage through different scopes.

Listing 2.8 shows a registered client able to use the client credentials grant type. Observe I have also configured a different scope. In this case, "CUSTOM" is just a name I chose, you can choose any name for the scopes. The name you choose should generally make the purpose of the scope easier to understand. For example, if this app needs to use the client credentials grant type to get a token to check the resource server's liveness state, then, maybe it's better to name the scope "LIVENESS" so it makes things obvious.

Listing 2.8 Configuring a registered client for the client credentials grant type

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
  RegisteredClient registeredClient =
    RegisteredClient.withId(UUID.randomUUID().toString())
      .clientId("client")
      .clientSecret("secret")
      .clientAuthenticationMethod(
         ClientAuthenticationMethod.CLIENT_SECRET_BASIC)    #A
      .authorizationGrantType(AuthorizationGrantType.CLIENT_CREDENTIALS)
      .scope("CUSTOM")    #B
      .build();

    return new InMemoryRegisteredClientRepository(registeredClient);
}
```

#A Allowing the registered client to use the client credentials grant type.
#B Configuring a scope to match the purpose for the access token request.

Figure 2.7 reminds you the client credentials flow we discussed in chapter 1. To get an access token, the client simply sends a request and authenticates using their credentials (the client ID and secret).

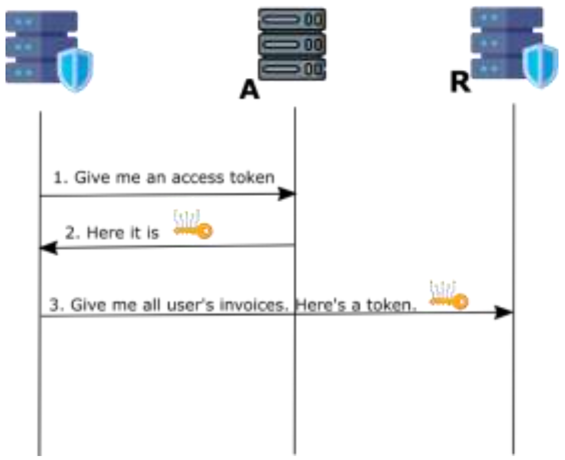

A
R
1. Give me an access token
2. Here it is
3. Give me all user's invoices. Here's a token.

Figure 2.7 The client credentials grant type. An app can get an access token by only authenticating with its client credentials

The following snippet shows a cURL token request. If you compare it with the request we used in section 2.2 when we ran the authorization code grant type, you'll observe this one is simpler. The client only needs to mention they use the client credentials grant type and the scope in which they request a token. The client uses its credentials with HTTP Basic on the request to authenticate.

```
curl -X POST
'http://localhost:8080/oauth2/token?grant_type=client_credentials&scope=CUS
TOM' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

The next snippet shows the HTTP response body containing the requested access token.

```
{
    "access_token": "eyJraWQiOiI4N2E3YjJiNS…",
    "scope": "CUSTOM",
    "token_type": "Bearer",
    "expires_in": 300
}
```

2.4 Using opaque tokens and introspection

By now, in this chapter, we have demonstrated the authorization code grant type (section 2.2) and the client credentials grant type (section 13.3). With both, we managed to configure clients that can get non-opaque access tokens. However, you can also easily configure the clients to use opaque tokens. In this section, I'll show you how to configure the registered clients to get opaque tokens and how the authorization server help with validating the opaque tokens.

Listing 2.9 shows how to configure a registered client to use opaque tokens. Remember that opaque tokes can be used with any grant type. In this section, I'll use the client credentials grant type to keep things simple and allow you to focus on the discussed subject. You can as well generate opaque tokens with the authorization code grant type.

Listing 2.9 Configuring clients to use opaque tokens

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
  RegisteredClient registeredClient =
    RegisteredClient.withId(UUID.randomUUID().toString())
        .clientId("client")
        .clientSecret("secret")
        .clientAuthenticationMethod(
           ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
        .authorizationGrantType(AuthorizationGrantType.CLIENT_CREDENTIALS)
        .tokenSettings(TokenSettings.builder()
           .accessTokenFormat(OAuth2TokenFormat.REFERENCE)      #A
            .build())
        .scope("CUSTOM")
        .build();

    return new InMemoryRegisteredClientRepository(registeredClient);
  }
```

#A Configuring the client to use opaque access tokens

If you request an access token as you learned in section 2.3, you'll get an opaque token. This token is shorter and doesn't contain data. The next snippet is a cURL request for an access token.

```
curl -X POST
'http://localhost:8080/oauth2/token?grant_type=client_credentials&scope=CUS
TOM' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

The next snippet shows the response similar to what you got in return when we were expecting non-opaque tokens. The only difference is the token itself which is no longer a JWT token, but an opaque one.

```
{
    "access_token": "iED8-...",
    "scope": "CUSTOM",
    "token_type": "Bearer",
    "expires_in": 299
}
```

The next snippet shows an example of a full opaque token. Observe is much shorter and doesn't have the same structure as a JWT (the three parts separated by dots).

```
iED8-
aUd5QLTfihDOTGUhKgKwzhJFzYWnGdpNT2UZWO3VVDqtMONNdozq1r9r7RiP0aNWgJipcEu5Hec
AJ75VyNJyNuj-kaJvjpWL5Ns7Ndb7Uh6DI6M1wMuUcUDEjJP
```

Since an opaque token doesn't contain data, how can someone validate it and get more details about the client (and potentially user) for whom the authorization server generated it? The easiest way (and most used) is directly asking the authorization server. The authorization server exposes an endpoint where one can send a request with the token. The authorization server replies with the needed details about the token. This process is called introspection (figure 2.8).

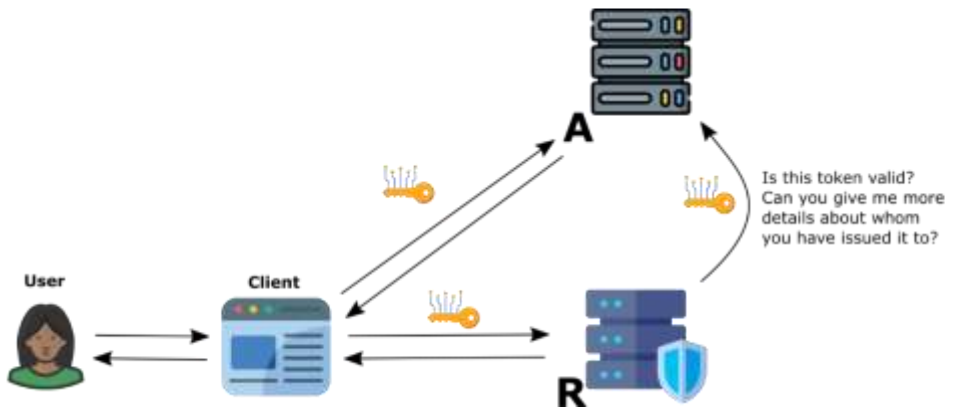

User
Client
A
R
Is this token valid? Can you give me more details about whom you have issued it to?

Figure 2.8 Token introspection. When using opaque tokens, the resource server needs to send requests to the authorization server to discover if the token is valid and more details about to whom it was issued.

The next snippet shows the cURL call to the introspection endpoint the authorization server exposes. The client must use HTTP Basic to authenticate using their credentials when sending the request. The client sends the token as a request parameter and receives details about the token in response.

```
curl -X POST 'http://localhost:8080/oauth2/introspect?token=iED8-…' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

The next snippet shows an example of a response to the introspection request for a valid token. When the token is valid, its status appears as "active" and the response provides all the details the authorization server has about the token.

```
{
    "active": true,
    "sub": "client",
    "aud": [
        "client"
    ],
    "nbf": 1682941720,
    "scope": "CUSTOM",
    "iss": "http://localhost:8080",
    "exp": 1682942020,
    "iat": 1682941720,
    "jti": "ff14b844-1627-4567-8657-bba04cac0370",
    "client_id": "client",
    "token_type": "Bearer"
}
```

If the token doesn't exist or has expired, its active status is false, as shown in the next snippet.

```
{
    "active": false,

}
```

The default active time for a token is 300 seconds. In examples, you'll prefer to make the token life longer. Otherwise, you won't have enough time to use the token for tests, which can become frustrating. Listing 2.10 shows you how to change the token time to live. I prefer to make it very large for example purposes (like 12 hours in this case), but remember never to configure it this large for a real-world app. In a real app, you'd usually go with a time to live from 10 to 30 minutes max.

Listing 2.10 Changing the access token time to live

```
RegisteredClient registeredClient = RegisteredClient
        .withId(UUID.randomUUID().toString())
        .clientId("client")
         // …
        .authorizationGrantType(AuthorizationGrantType.CLIENT_CREDENTIALS)
        .tokenSettings(TokenSettings.builder()
            .accessTokenFormat(OAuth2TokenFormat.REFERENCE)
            .accessTokenTimeToLive(Duration.ofHours(12))    #A
            .build())
        .scope("CUSTOM")
        .build();
```

#A Setting 12 hours as the access token time to live.

2.5 Revoking tokens

Suppose you discover a token has been stolen. How could you make a token invalid for use? Token revocation is a way to invalidate a token the authorization server previously issued. Normally, an access token lifespan is short, so stealing a token still makes it difficult for one to use it. But sometimes you want to be extra cautious.

The following snippet shows a cURL command you can use to send a request to the token revocation endpoint the authorization server exposes. You can use any of the projects we worked on in this chapter for the test. The revocation feature is active by default in a Spring Security authorization server. The request only requires the token that you want to revoke and HTTP Basic authentication with the client credentials. Once you send the request, the token cannot be used anymore.

```
curl -X POST 'http://localhost:8080/oauth2/revoke?token=N7BruErWm-44-…' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

If you use the introspection endpoint with a token you have revoked, you should observe the token is no longer active after revocation (even if its time to live hasn't expired yet).

```
curl -X POST 'http://localhost:8080/oauth2/introspect?token=N7BruErWm-44-…'
\
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

Using token revocation makes sense sometimes, but it's not something you'd always desire. Remember that if you want to use the revocation feature, this also implies you need to use introspection (even for non-opaque tokens) with every call to validate that the token is still active. Using introspection so often might have a big impact on the performance. You should always ask yourself: Do I really need this extra protection layer?

Remember our discussion in the first chapter of the book. Sometimes, hiding the key under the rug is enough, other times you need advanced, complex, and expensive alarm systems. What you use depends on what you protect.

2.6 Summary

- The Spring Security authorization server framework helps you build a custom OAuth 2/OpenID Connect authorization server from scratch.
- Since the authorization server manages the users' and clients' details, you must implement components defining how the app collects this data.
  - To manage the user details, the authorization server needs a similar Spring Security component like any other web app: an implementation of a `UserDetailsService`.
  - To manage the client details, the authorization server provides another contract you must implement: the `RegisteredClientRepository`.
- You can register clients that use various authentication flows (grant types). Preferably, the same client shouldn't use both user dependent (like the authorization code grant type) and user-independent (like client credentials grant type) flows.
- When using non-opaque tokens (usually JWTs), you must also configure a component to manage the key pairs the authorization server uses to sign the tokens. This component is named the `JWKSource`.
- When using opaque tokens (tokens that don't contain data), the resource server must use the introspection endpoint to verify a token's validity and collect data necessary for authorization.
- Sometimes, you'd need a way to invalidate already issued tokens. The authorization server offers the revocation endpoint for this capability. When using revocation, the resource server must always introspect the tokens (even non-opaque ones) to verify their validity.

3 Implementing an OAuth 2 resource server

This chapter covers

- Implementing a Spring Security OAuth 2 resource server
- Using JWT tokens with custom claims
- Configuring introspection for opaque tokens or revocation
- Implementing more complex scenarios and multitenancy

In this chapter, we discuss securing a backend application in an OAuth 2 system. What we call a resource server in OAuth 2 terminology is actually simply a backend service. While in chapter 14 you learned how to implement the authorization server responsibility using Spring Security, it's now time to discuss using the token the authorization server generates.

In real-world scenarios, you might or might not implement a custom authorization server as we did in chapter 2. Your organization might choose to use a third-party implementation instead of creating custom software. You can find many alternatives out there ranging from open-source solutions such as Keycloak to enterprise products such as Okta, Cognito, or Azure AD. An example with Keycloak you can actually find in chapter 18 of the first edition of Spring Security in Action.

While you have options to configure an authorization server without needing to implement your own, you'll have to properly implement the authentication and authorization on your backend. For that reason, I think this chapter is essential; the skills you learn by reading it have a high chance of helping you with your work. Figure 3.1 reminds you about the OAuth 2 actors and where we are with our learning plan for this book part.

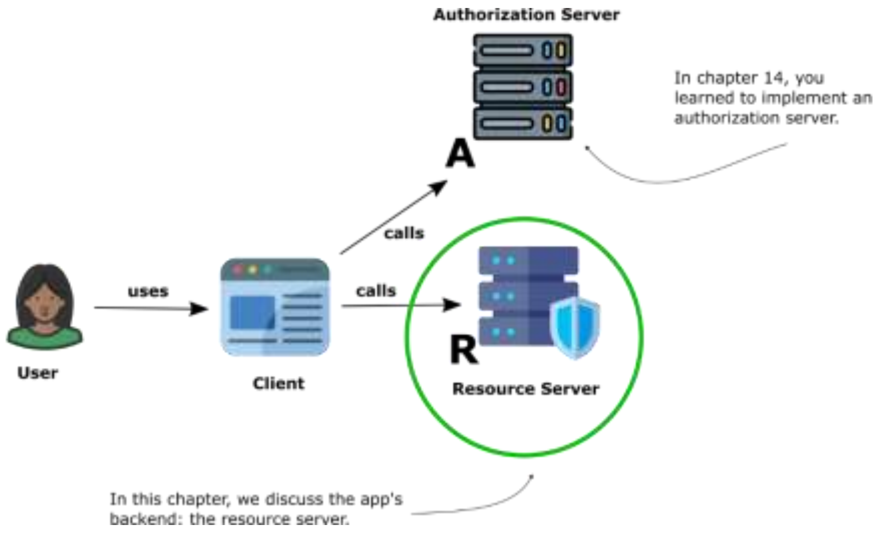

Authorization Server
A
In chapter 14, you learned to implement an authorization server.
calls
uses
calls
User
Client
R
Resource Server
In this chapter, we discuss the app's backend: the resource server.

Figure 3.1 In OAuth 2 the app's backend is called a resource server because it protects the users' and clients' resources (data and actions that can be done on data).

We'll begin this chapter in section 3.1 with discussing the resource server configuration for JWTs. You'll most often find JWT used with the OAuth 2 system today; that's why we also start with them. In section 3.2 we discuss customizing the JWT tokens and using the custom values in its body or header claims.

In section 3.3 we discuss configuring the resource server to use introspection for validating the tokens. The introspection process is useful when using opaque tokens or when you want your system to be able to revoke tokens before their expiration date.

We'll end the chapter discussing more advanced configuration cases like multitenancy in section 3.4.

3.1 Configuring JWT validation

In this section, we discuss configuring a resource server to validate and use JWT tokens. JWT tokens are non-opaque tokens (they contain data the resource server uses for authorization). To use JWT tokens, the resource server will need to prove they are authentic, meaning that the expected authorization server has indeed issued them as a proof of authentication of a user and/or a client. Secondly, the resource server needs to read the data in the token and use it to implement authorization rules.

We'll learn to configure a resource server by putting it "in action" – implementing one and configuring it from scratch. We'll start by creating a new Spring Boot project and add the needed dependencies. We'll then implement a demo endpoint (a resource to use for test purposes) and work on the configuration for authentication and authorization. Here're the steps we'll follow:

1. Add the needed dependencies to the project (in the pom.xml file since we use Maven).
2. Declare a dummy endpoint that we'll use to test our implementation.
3. Implement authentication for JWT tokens by configuring the service with the public key set URI.
4. Implement the authorization rules.
5. Test the implementation by
   a. Generating a token with the authorization server
   b. Using the token to call the dummy endpoint we created in step 2.

Listing 3.1 presents the needed dependencies. Aside from the web and Spring Security dependencies, we'll also add the resource server starter.

Listing 3.1 Dependencies for implementing a resource server

```
<dependency>    #A
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

#A The resource server starter provides the needed dependencies for implementing an app as an OAuth 2 resource server.

Once we have the dependencies in place, we create a dummy endpoint that we'll use to test our implementation at the end. Listing 3.2 presents a simple controller which exposes an endpoint at path /demo.

Listing 3.2 Declaring a simple endpoint for test purposes

```
@RestController
public class DemoController {

  @GetMapping("/demo")    #A
  public String demo() {
    return "Demo";
  }
}
```

#A Definition of the dummy endpoint we need to test our configurations after finishing our implementation.

For this example, you need to use an authorization server.

Since we want to start both the authorization server and the resource server simultaneously on the same system, we'll need to configure different ports for them. Since the authorization server has the default port 8080, we can change the resource server port to another one. I changed it to 9090, but you can use any free port on your system. The next code snippet shows the property to add to your application.properties file to change the port.

```
server.port=9090
```

Remember from chapter 2 that an OpenID Connect authorization server exposes a URL you can use to get its configuration (including the URL for authorization, token, public key set and others). The next snippet presents the so-called "well-known" URL.

```
http://localhost:8080/.well-known/openid-configuration
```

You need this link to get information about the URL that the authorization server exposes to provide the public key set that the resource server can use to validate tokens. The resource server needs to call this endpoint and get the set of public keys. Then, the resource server uses one of these keys to validate the access token's signature (figure 3.2).

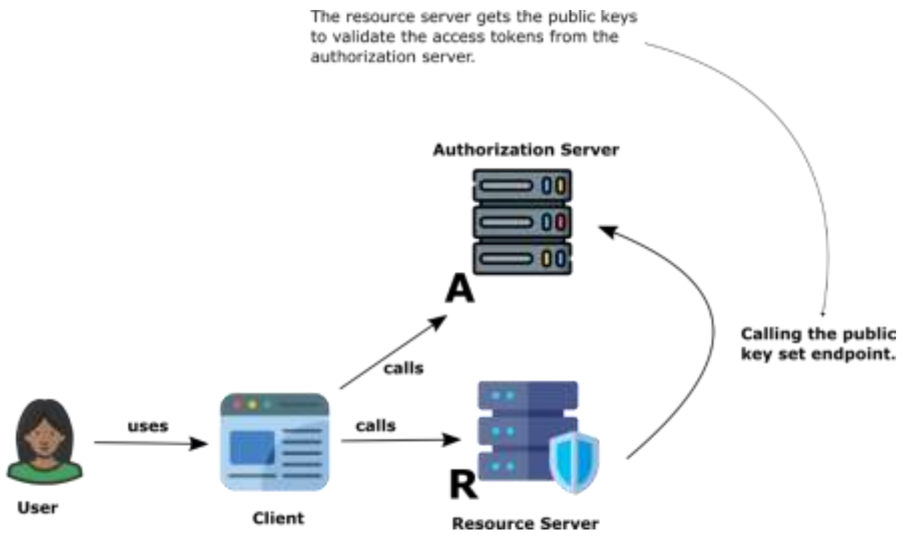

The resource server gets the public keys to validate the access tokens from the authorization server.
Authorization Server
A
Calling the public key set endpoint.
calls
uses
calls
User
Client
R
Resource Server

Figure 3.2 The resource server gets the public key set from the authorization server using an endpoint that the authorization server exposes. The resource server uses these keys to validate the access token's signature.

Listing 3.3 reminds you about the response you get when calling the well-known configuration endpoint the authorization server exposes. As you can observe, the public key set URI is among the other data provided. The public key set URI is what we need to configure in the resource server so it can validate the JWT tokens.

Listing 3.3 The response of well-known OpenID configuration contains the key set URI

```
{
    "issuer": "http://localhost:8080",
    "authorization_endpoint": "http://localhost:8080/oauth2/authorize",
    "device_authorization_endpoint":
 [CA]"http://localhost:8080/oauth2/device_authorization",
    "token_endpoint": "http://localhost:8080/oauth2/token",

    …

    "jwks_uri": "http://localhost:8080/oauth2/jwks",     #A
    …

}
```

#A The key set endpoint provides the public parts of the asynchronous key pairs configured on the authorization server side. The authorization server uses the private parts to sign the tokens. The resource server can use the public parts to validate the tokens.

To configure the public key set URI, we'll first declare it in the project's application.properties file. The configuration class can inject it into an attribute field and then use it to configure the resource server authentication.

```
keySetURI=http://localhost:8080/oauth2/jwks
```

Listing 3.4 shows the configuration class injecting the public key set URI value into an attribute. The configuration class also defines a bean of the type `SecurityFilterChain`. The application will use the `SecurityFilterChain` bean to configure the authentication, similar to what we did in all this book's previous chapters.

Listing 3.4 Injecting the property value in the configuration class

```
@Configuration
public class ProjectConfig {

  @Value("${keySetURI}")     #A
  private String keySetUri;

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {


    return http.build();
  }
}
```

#A Inject the key set URI value in an attribute of the configuration file. You'll need it for the filter chain configuration.

To configure the authentication, we'll use the `oauth2ResourceServer()` method of the `HttpSecurity` object. This method is similar to `httpBasic()` and `formLogin()`, which we used before in the second and third parts of this book.

Similar to `httpBasic()` and `formLogin()`, you need to provide an implementation of the `Customizer` interface to configure the authentication. In listing 3.5, you can observe how I used the `jwt()` method of the `Customizer` object to configure JWT authentication. Then, I used a `Customizer` on the `jwt()` method to configure the public key set URI (using the `jwkSetUri()` method).

Listing 3.5 Configuring the authentication with JWT tokens

```
@Configuration
public class ProjectConfig {
    @Value("${keySetURI}")
    private String keySetUri;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
      throws Exception {

      http.oauth2ResourceServer(      #A
        c -> c.jwt(      #B
           j -> j.jwkSetUri(keySetUri)      #C
        )
      );

      return http.build();
    }
  }
```

#A Configuring the app as an OAuth 2 resource server.
#B Configuring the resource server to use JWT tokens for authentication.
#C Configuring the public key set URL that the resource server will use to validate the tokens.

Remember to make the endpoints require authentication. By default the endpoints aren't protected, so to test the authentication, you first need to make sure your /demo endpoint requires authentication. The following code snippet configures the app's authorization rules. For this example, we can simply configure all endpoint to require authentication.

```
http.authorizeHttpRequests(
    c -> c.anyRequest().authenticated()
);
```

Listing 3.6 presents the full content of the configuration class.

Listing 3.6 The full configuration class

```
@Configuration
public class ProjectConfig {

  @Value("${keySetURI}")
  private String keySetUri;

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {


    http.oauth2ResourceServer(
      c -> c.jwt(
         j -> j.jwkSetUri(keySetUri)
      )
    );

    http.authorizeHttpRequests(
      c -> c.anyRequest().authenticated()
    );

    return http.build();
  }
}
```

Now you should start the resource server application you just created. Make sure that your authorization server is still up and running. You'll need to use the skills you learned in chapter 2 to generate an access token. Let's remember together the steps for the authorization code grant type (but remember you could get the token using any other grant type – it's irrelevant for the resource server how you got the access token as long as you have one).

The steps you need to follow with the authorization code grant type are the following (figure 3.3):

6. Redirect the user to login at the authorization server /authorize endpoint.
7. Use the user's credentials to authenticate – the authorization server will redirect you to the redirect URI and provide the authorization code.
8. Take the authorization code provided after the redirect and use the /token endpoint to request a new access token.

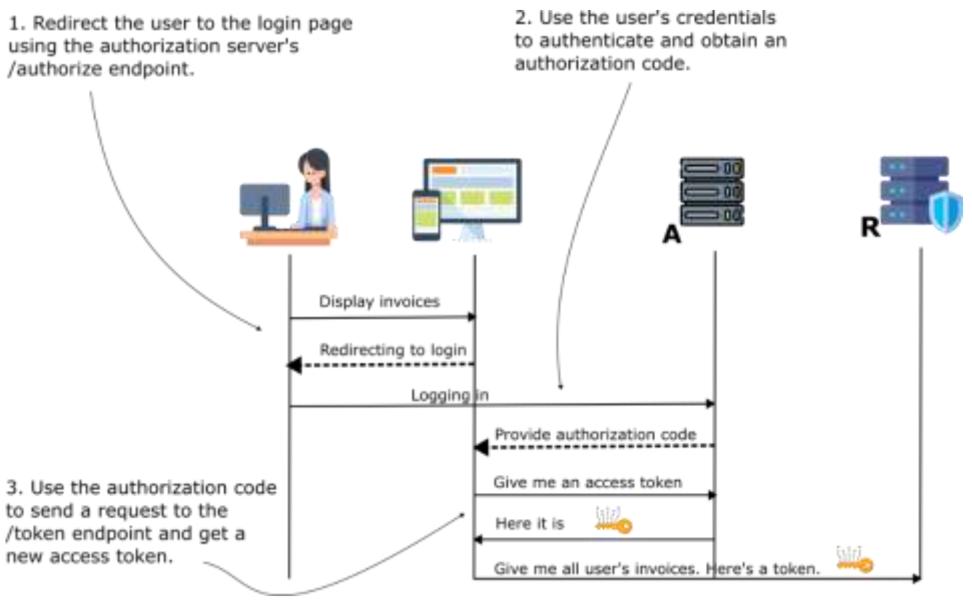

1. Redirect the user to the login page using the authorization server's /authorize endpoint.
2. Use the user's credentials to authenticate and obtain an authorization code.
A
R
Display invoices
Redirecting to login
Logging in
Provide authorization code
Give me an access token
Here it is
Give me all user's invoices. Here's a token.
3. Use the authorization code to send a request to the /token endpoint and get a new access token.

Figure 3.3 The authorization code grant type. The client redirects the user to the authorization server's login page. After the user successfully authenticates, the authorization server redirects back to the client, providing an authorization code. The client uses the authorization code to get an access token.

The next snippet (right after the bullet points) shows the URL you can use in your browser to redirect to the authorization server's /authorize endpoint. Remember you need to provide a few parameters, and their values should comply with what you configured in the authorization server. The parameters you must send are

- *response_type* – use value "code" if you want to use the authorization code grant type.

- *client_id* – the client ID.
- *scope* – the scope you want to access. It can be any of the scopes configured within the authorization server.
- *redirect_uri* – the URI to which the authorization server redirects the client after successful authentication. The redirect URI should be one of those configured in the authorization server.
- *code_challenge* – If using PKCE (Proof Key for Code Exchange) you need to provide the code challenge from the code challenge and verifier pair.
- *code_challenge_method* - If using PKCE, you must specify the hash function you used to encrypt the code verifier (for example, SHA-256).

```
http://localhost:8080/oauth2/authorize?response_type=code&client_id=client&
scope=openid&redirect_uri=https://www.manning.com/authorized&code_challenge
=QYPAZ5NU8yvtlQ9erXrUYR-T5AGCjCF47vN-KsaI2A8&code_challenge_method=S256
```

NOTE Remember that you must paste the authorization URL in a browser's address bar to send the request.

Log in using the valid user credentials configured in the authorization server and then wait to be redirected to the requested redirect URI. The authorization server provides the authorization code which you must use in the request to the /token endpoint.

The next snippet shows an example of cURL command sending a request to the /token endpoint to get an access token. Mind that I have truncated the authorization code value to fit in the page.

```
curl -X POST 'http://localhost:8080/oauth2/token? \
client_id=client& \
redirect_uri=https://www.manning.com/authorized& \
grant_type=authorization_code& \
code=IhKRpq7GJ7P5VQI_...& \
code_verifier=qPsH306-ZDDaOE8DFzVn05TkN3ZZoVmI_6x4LsVglQI' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

The following snippet presents a response body for the /token request. I have as well truncated the token values in the snippet.

```
{
    "access_token": "eyJraWQiOiI2Zjk5ZmE3MC…",
    "scope": "openid",
    "id_token": "eyJraWQiOiI2Zjk5ZmE3MC0xNTQ2LTRkMjM…",
    "token_type": "Bearer",
    "expires_in": 299
}
```

You can use the access token now when calling any endpoint that requires authentication. The following snippet shows a cURL command to send a request to the /demo endpoint. Observe that the access token must be sent in the `Authorization` header using the `Bearer` prefix (figure 3.4). The Bearer prefix implies that the one having the access token value can use it in the same way as any other party having it.

Figure 3.4 Analogy to *The Lord of the Rings* novel by J.R.R. Tolkien. The access token is a precious resource. It gives access to several resources to whomever possesses it.

Snippet below shows the cURL command you can use to send a request to the /demo endpoint using the access token from the authorization server.

```
curl 'http://localhost:9090/demo' \
--header 'Authorization: Bearer eyJraW…'
```

3.2 Using customized JWT tokens

Systems' needs are different from one another, even in regard to authentication and authorization. Often, it happens you need to transfer custom values between the authorization server and the resource server through the access token. The resource server can use such values to apply various authorization rules.

In this section, we'll implement an example where the authorization and resource servers use custom claims in the access token. The authorization server customizes the JWT adding a claim named "priority" in the JWT. The resource server reads the "priority" claim and adds its value to the authentication instance in the security context. From there, the resource server can use it when implementing any authorization rule.

We'll follow these steps:

1. Change the authorization server to add the custom claim to the access token.
2. Change the resource server to read the custom claim and store it in the security context.
3. Implement an authorization rule that uses the custom claim.

First step first! We need to add a custom value in the access token's body to the `SecurityConfig` class. In the authorization server, you do this by adding a bean of type `OAuth2TokenCustomizer`. The next code snippet demonstrates such a bean's definition. To make things simple and allow you to focus on the example, I chose to add a dummy value in a field I named “priority”. In real-world apps, such custom fields would have a purpose, and you’ll potentially have to write certain logic for setting their value.

```
@Bean
public OAuth2TokenCustomizer<JwtEncodingContext> jwtCustomizer() {
  return context -> {
    JwtClaimsSet.Builder claims = context.getClaims();
    claims.claim("priority", "HIGH");
  };
}
```

With this minimal change, the access tokens now contain a custom “priority” field. The next snippet shows a JWT access token I generated in its Base64 encoded format, and listing 3.7 shows the decoded body where you observe the priority field.

```
eyJraWQiOiI5ZTBjOTQ5Ny0zYmMyLTQ4Y2YtODU5MC04N2JmZjE2ZjczOTAiLCJhbGciOiJSUzI
1NiJ9.eyJzdWIiOiJiaWxsIiwiYXVkIjoiY2xpZW50IiwibmJmIjoxNjg3MjYzMzI5LCJzY29wZ
SI6WyJvcGVuaWQiXSwiaXNzIjoiaHR0cDovL2xvY2FsaG9zdDo4MDgwIiwiZXhwIjoxNjg3MjYz
NjI5LCJwcmlvcml0eSI6IkhJR0giLCJpYXQiOjE2ODcyNjMzMjl9.HrQECSO17tZD8HKXP0U7gm
dmea01vPgVypvcf3oR3uawiMdI_joQBsLY0zNWBIgktKn2w9-
rvgtjD2xmhWZgSxRsDW_GZofqOzV9T-
5llMuZlakF7SQLyI67UJZKuPTJK8hBd1OhnurGo7ikPfDWhaqyychKu_uI7SdFrQQVgVqbrmHii
syoURIrI9EwOhB036M7UPJnIWtOWc34fAoFHxqhPuGIVesHHX5qm6wx-
8_Orjz96eOujVSEuUGRNVtz35_SRjhozcLzgIo3Rt9lUfLI7HSzulfXTCpxtxja-
1E_l_dsk4VHSvLYJUZjlERp5kVJqSO_keaJt8JbDQ0new
```

Listing 3.7 shows the decoded body of the previously presented access token. Remember you can easily use the jwt.io tool to get the decoded form of a JWT. Alternatively, you can individually Base64 decode either the header or the body of the access token using any other Base64 decoder. Listing 3.7 demonstrates our changes on the authorization server work correctly.

Listing 3.7 The Base64 decoded body of the customized JWT access token

```
{
  "sub": "bill",
  "aud": "client",
  "nbf": 1687263329,
  "scope": [
    "openid"
  ],
  "iss": "http://localhost:8080",
  "exp": 1687263629,
  "priority": "HIGH",    #A
  "iat": 1687263329
}
```

#A The custom “priority” claim we added to the access token.

As a second step, we make the changes on the resource server. You can continue working on the example we used in section 3.1, but to make your learning easier, I created a separate project for this example.

The list of steps we need to follow to make our resource server understands the custom claims in the access token is the following:

a. Create a custom authentication object – this object will define the new shape, which will include the custom data.

b. Create a JWT authentication converter object – this object will define the logic for translating the JWT into the custom authentication object.

c. Configure the JWT authentication converter you created at step "b" to be used by the authentication mechanism.

d. Change the /demo endpoint to return the authentication object from the security context.

e. Test the endpoint and check that the authentication object contains the custom "priority" field.

Listing 3.8 presents the definition of the authentication object. The authentication object should be any class that inherits directly or indirectly the `AbstractAuthenticationToken` class. Since we use JWTs, it's more comfortable to extend the more specific `JwtAuthenticationToken`. This way you'll extend directly the casual shape of an authentication object as designed for JWT access tokens.

Observe that the customization in listing 3.8 adds a field named "priority". This field will hold the value from the custom claim in the access token's body. In a similar way, you can add any other custom details your app might need for authorization purposes. Having these details directly in the authentication object from the security context makes configurations easy to write regardless of whether we choose to apply them at the endpoint level or method level.

Listing 3.8 Defining a custom authentication object

```
public class CustomAuthentication
  extends JwtAuthenticationToken {    #A

  private final String priority;     #B

  public CustomAuthentication(
    Jwt jwt,
    Collection<? extends GrantedAuthority> authorities,
    String priority) {

    super(jwt, authorities);
    this.priority = priority;
  }

  public String getPriority() {
    return priority;
  }
}
```

#A Customizing the authentication object by extending the JwtAuthenticationToken class.
#B Adding the custom field "priority"

Having a custom shape of the authentication object, the next thing you need to do is instruct your app on how to translate the JWT into this custom object. You can do this by configuring a specific `Converter` as presented in listing 3.9. Observe the two generic types we used, the `Jwt` and the `CustomAuthentication`. The first generic type, the `Jwt`, is the input for the converter, while the second type, the `CustomAuthentication` is the output. So what this converter does is change a `Jwt` object (which is a standard contract in Spring Security on how the JWT access token is read) into the custom type we implemented in listing 3.8.

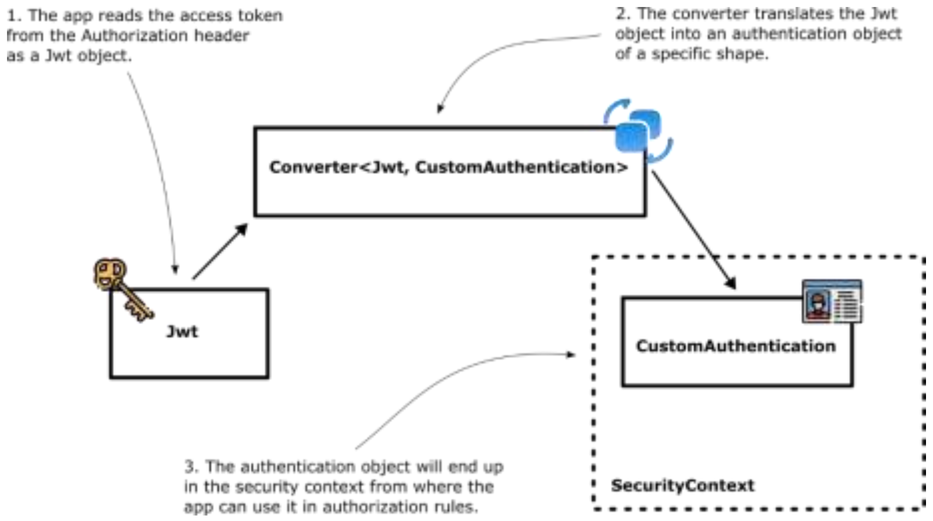

1. The app reads the access token from the Authorization header as a Jwt object.
2. The converter translates the Jwt object into an authentication object of a specific shape.
Converter<Jwt, CustomAuthentication>
Jwt
CustomAuthentication
3. The authentication object will end up in the security context from where the app can use it in authorization rules.
SecurityContext

Figure 3.5 A custom converter implements logic to put information from the access token in a custom authentication shape.

Listing 3.9 Converting the access token to an authentication object

```
@Component
public class JwtAuthenticationConverter
  implements Converter<Jwt, CustomAuthentication> {

  @Override
  public CustomAuthentication convert(Jwt source) {
    List<GrantedAuthority> authorities =
       List.of(() -> "read");

    String priority =      #A
       String.valueOf(source.getClaims().get("priority"));

    return new CustomAuthentication(source,
                                    authorities,
                                    priority);    #B
  }
}
```

#A Getting the priority values from the token's custom claim.
#B Setting the priority value in the authentication object.

You may also observe in listing 3.9 that I have defined a dummy authority. In a real scenario, you'd either take these from the access token (considering they're managed at the authorization server level), or from a database or other third-party system (considering they're managed from a business-level point of view. In this case, I simplified the example and simply added a dummy "read" authority for all requests. But it's important to remember that this is also the place where you'd deal with authorities (which should also end up in the authentication object from the security context since they're essential details for authorization rules in most cases).

In listing 3.10 you find out how to configure the custom converter. In this case, I used dependency injection to get the converter bean from the Spring context. Then, I used the `jwtAuthenticationConverter()` method of the JWT authentication configurer.

Listing 3.10 Configuring the custom authentication converter

```
@Configuration
public class ProjectConfig {

  // omitted code

  private final JwtAuthenticationConverter converter;    #A

  // omitted constructor

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {

    http.oauth2ResourceServer(
       c -> c.jwt(
          j -> j.jwkSetUri(keySetUri)
                .jwtAuthenticationConverter(converter)    #B
          )
    );

    http.authorizeHttpRequests(
      c -> c.anyRequest().authenticated()
    );

    return http.build();
  }
}
```

#A Injecting the converter object in a class field.
#B Configuring the converter object within the authentication mechanism.

That's all about the configuration we needed to implement to use the access token's custom claim. Let's test our implementation and prove that it works the way we expect. The next code snippet shows the changes I made to the /demo endpoint. I made the /demo endpoint return the authentication instance from the security context. Since Spring knows to inject the value automatically in a parameter of type `Authentication`, I just needed to add this parameter and then make the endpoint's action method return it as-is.

```
@GetMapping("/demo")
public Authentication demo(Authentication a) {
    return a;
  }
```

If everything works as desired, when sending a request to the /demo endpoint you'll get a response with a body similar to the one presented in listing 3.11. Observe that the custom "priority" attribute correctly appears in the authentication object having the value "HIGH".

Listing 3.11 The /demo endpoint response contains the priority field

```
  {
    "authorities": [
      {
        "authority": "read"
      }
    ],
    "details": {
       "remoteAddress": "0:0:0:0:0:0:0:1",
       "sessionId": null
    },
    "authenticated": true,
      …

    "name": "bill",
    "priority": "HIGH",     #A
   }
```

#A The custom claim value appears in the authentication instance.

3.3 Configuring token validation through introspection

In this section, we discuss using introspection for access token validation. If your app uses opaque tokens or if you want a system where you can revoke tokens at the authorization server level, then introspection is the process you must use for token validation. Figure 3.6 reminds you of the introspection process, which we discussed in detail in section 14.4.

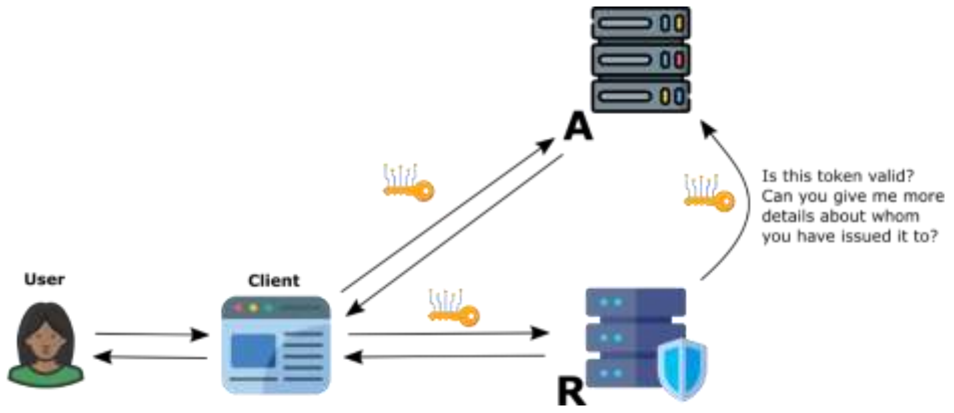


Figure 3.6 Token introspection. When the resource server can't rely on validating access tokens based on their signature (need for token revocation) or where details are not transferred within the token (opaque tokens), the resource server needs to send requests to the authorization server to find out whether a token is valid and get more details about it.

We'll implement a resource server to demonstrate the use of introspection. To achieve our goal, we must follow these steps:

1. Make sure that the authorization server recognizes the resource server as a client. The resource server needs client credentials registered on the authorization server side.
2. Configure authentication on the resource server side to use introspection.
3. Obtain an access token from the authorization server.
4. Use a demo endpoint to prove that the configuration works the way we expect with the access token we got in step 3.

The next code snippet shows you an example of creating a client instance which we'll register at the authorization server side. This client represents our resource server. As you observe from figure 3.6, the resource server sends requests to the authorization server (for introspection) so, this way, it becomes also a client of the authorization server.

To send the introspection requests, the resource server needs client credentials to authenticate, similar to any other client.

```
RegisteredClient resourceServer =
   RegisteredClient.withId(UUID.randomUUID().toString())
             .clientId("resource_server")
             .clientSecret("resource_server_secret")
             .clientAuthenticationMethod(
                ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
             .authorizationGrantType(
                AuthorizationGrantType.CLIENT_CREDENTIALS)
             .build();
```

I remind you that passwords and any configuration data should never be hardcoded as I did in the previous snippet. I simplify these examples as much as possible to allow you focus on the subject we discuss. In a real-world app, you should put the configurations in files outside of the implementation and your should securely persist secret details (like credentials) somewhere.

Listing 3.12 shows you how to add both instances of client details (the client's and the resource server's) to the authorization server's `RegisteredClientRepository` component.

Listing 3.12 The RegisteredClientRepository definition

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
  RegisteredClient registeredClient =      #A
    RegisteredClient.withId(UUID.randomUUID().toString())
      .clientId("client")
      .clientSecret("secret")
      .clientAuthenticationMethod(
         ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
      .authorizationGrantType(AuthorizationGrantType.CLIENT_CREDENTIALS)
      .tokenSettings(TokenSettings.builder()
          .accessTokenFormat(OAuth2TokenFormat.REFERENCE)
          .accessTokenTimeToLive(Duration.ofHours(12))
```

```
            .build())
        .scope("CUSTOM")
        .build();

    RegisteredClient resourceServer =      #B
      RegisteredClient.withId(UUID.randomUUID().toString())
        .clientId("resource_server")
        .clientSecret("resource_server_secret")
        .clientAuthenticationMethod(
           ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
        .authorizationGrantType(
           AuthorizationGrantType.CLIENT_CREDENTIALS)
        .build();

      return new InMemoryRegisteredClientRepository(
                      registeredClient,       #C
                      resourceServer);
    }
```

#A Defining a client details instance for the client app.
#B Defining a client details instance for the resource server (which also becomes a client when calling the introspection endpoint).
#C Adding both client details instances in the authorization server's repository.

With the changes made in listing 3.12, we now have a set of credentials our resource server can use to call the introspection endpoint that the authorization server exposes. We can start implementing the resource server. Listing 3.13 shows how I configured in the properties files the three essential values needed for introspection:

- The introspection URI that the authorization server exposes allows the resource server to validate tokens.
- The resource server client ID allows the resource server to identify itself when calling the introspection endpoint.
- The resource server client secret that the resource server uses together with its client ID to authenticate when sending requests to the introspection endpoint.

Along with these, I also changed the server port to 9009, a different one than the application server's (8080) this way allowing both apps run simultaneously.

Listing 3.13 The resource server application.properties file

```
server.port=9090    #A

introspectionUri=http://localhost:8080/oauth2/introspect    #B
resourceserver.clientID=resource_server    #C
resourceserver.secret=resource_server_secret     #D
```

#A Changing the resource server's port to allow both the resource server and authorization server to run simultaneously.
#B Configure the introspection URI as a property.
#C Configure the resource server client ID as a property.
#D Configure the resource server client secret as a property.

You can then inject the values in the properties file in fields of the configuration class and use them to set up the authentication. Listing 3.14 shows the configuration class injecting the values from the properties file into fields.

Listing 3.14 Injecting the values in fields of the configuration class

```
@Configuration
public class ProjectConfig {

  @Value("${introspectionUri}")    #A
  private String introspectionUri;

  @Value("${resourceserver.clientID}")     #A
  private String resourceServerClientID;

  @Value("${resourceserver.secret}")    #A
  private String resourceServerSecret;


}
```

#A Injecting the introspection URI, introspection client ID and introspection secret from the properties file in fields of the configuration class.

Use the introspection URI and the credentials to configure the authentication. You configure the authentication similarly to how we configured it for JWT access tokens – we use the `oauth2ResourceServer()` method of the `HttpSecurity` object. However, we call a different configuration method of the `oauth2ResourceServer()` customizer object: `opaqueToken()`. For the `opaqueToken()` method, we configure the introspection URI and credentials. Listing 3.15 presents all this setup.

Listing 3.15 Configuring the resource server authentication for opaque tokens

```
@Configuration
public class ProjectConfig {

  @Value("${introspectionUri}")
  private String introspectionUri;

  @Value("${resourceserver.clientID}")
  private String resourceServerClientID;

  @Value("${resourceserver.secret}")
  private String resourceServerSecret;

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {

     http.oauth2ResourceServer(
        c -> c.opaqueToken(    #A
          o -> o.introspectionUri(introspectionUri)      #B
                .introspectionClientCredentials(     #C
                    resourceServerClientID,
                    resourceServerSecret)
             )
     );



     return http.build();
    }
  }
```

#A Configuring resource server authentication for opaque tokens.
#B Configuring the introspection URI the resource server should use to validate and get details about tokens.
#C Configuring the credentials the resource server must use to authenticate when calling the authorization server's introspection URI.

Remember to add also the authorization configurations. The next code snippet shows you the standard way you learned to make all the endpoints require requests to be authenticated.

```
http.authorizeHttpRequests(
  c -> c.anyRequest().authenticated()
);
```

Listing 3.16 shows the full content of the configuration class.

Listing 3.16 Full content of the configuration class

```
@Configuration
public class ProjectConfig {

  @Value("${introspectionUri}")
  private String introspectionUri;

  @Value("${resourceserver.clientID}")
  private String resourceServerClientID;

  @Value("${resourceserver.secret}")
  private String resourceServerSecret;

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {

     http.oauth2ResourceServer(
        c -> c.opaqueToken(
          o -> o.introspectionUri(introspectionUri)
                .introspectionClientCredentials(
                    resourceServerClientID,
                    resourceServerSecret)
            )
     );

     http.authorizeHttpRequests(     #A
        c -> c.anyRequest().authenticated()
     );

     return http.build();
  }
}
```

#A Adding the endpoint authorization configuration. Requests for any endpoint need authentication.

A simple /demo endpoint like the one the next code snippet presents is enough for us to test that authentication works correctly.

```
@RestController
public class DemoController {
    @GetMapping("/demo")
    public String demo() {
      return "Demo";
    }
  }
```

You can start now both applications: the authorization server and the resource server. Both should run simultaneously. In the next code snippet, you find the cURL command you can use to send a request to the /token endpoint. To simplify this example, I use the client credentials grant type, but you could use any grant type you learned in chapter 2 to get the access token. Remember that the resource server configuration is the same regardless of how you get the access token.

```
curl -X POST 'http://localhost:8080/oauth2/token? \
client_id=client& \
grant_type=client_credentials' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

If the request is successful, you’ll get the access token back in response. The response body looks like the next snippet. I have truncated the token’s value to make it fit better in page.

```
{
    "access_token": "2zLyYA8b6Q54-…",
    "token_type": "Bearer",
    "expires_in": 43199
}
```

Same as for the JWT access tokens, when sending a request to a protected endpoint, add the token as the value of the “Authorization” header. The access token value must be prefixed with the string “Bearer”. The next snippet shows the cURL command you can use to send a request to the /demo endpoint. If everything works correctly, you’ll get back the “Demo” string in the body in a 200 OK response status.

```
curl 'http://localhost:9090/demo' \
--header 'Authorization: Bearer 2zLyYA8b6Q54-…'
```

3.4 Implementing multitenant systems

In real-world apps, things aren’t always perfect. Sometimes we’re in a situation where we have to adapt our implementation to match some non-standard case when integrating with a third party. Also, sometimes we need to implement backends that rely on multiple authorization servers for authentication and authorization (multitenant systems). How should we implement our apps’ configurations in such cases?

Fortunately, Spring Security offers flexibility for implementing any scenario. In this section, we discuss implementing resource server configurations for more complex cases, such as multitenant systems or interacting with apps that don’t follow the standards.

Let’s remember Spring Security’s authentication design that we discussed in detail the first two parts of this book. A filter intercepts the HTTP request. The authentication responsibility is then delegated to an authentication manager. The authentication manager further uses an authentication provider which implements the authentication logic.

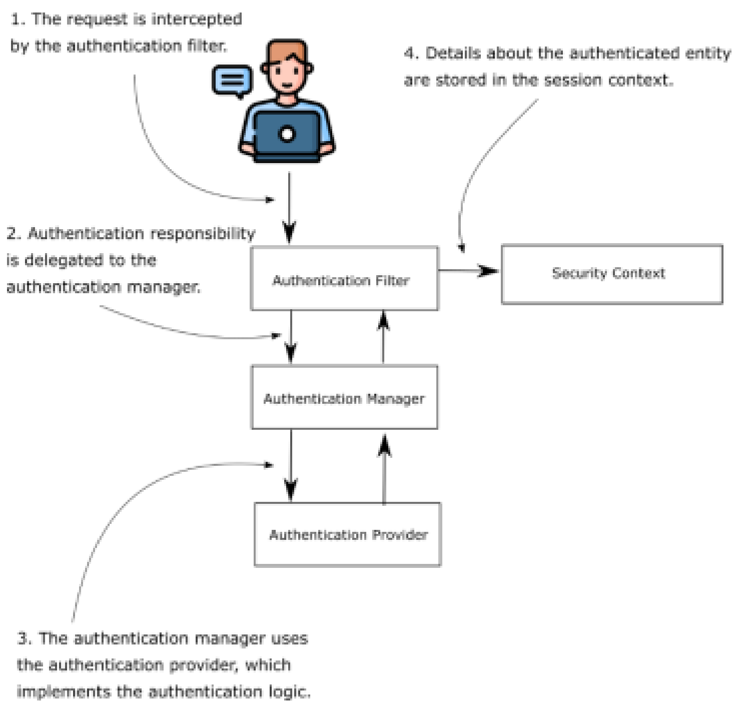

1. The request is intercepted by the authentication filter.
4. Details about the authenticated entity are stored in the session context.
2. Authentication responsibility is delegated to the authentication manager.
Authentication Filter
Security Context
Authentication Manager
Authentication Provider
3. The authentication manager uses the authentication provider, which implements the authentication logic.

Figure 3.7 The authentication class design. To fulfill the authentication process, the filter intercepts the request and then delegates it to an authentication manager component. The authentication manager further uses an authentication provider which implements the authentication logic. Once the authentication successfully fulfils the app stores the details about the authentication principal in the security context.

Why is it important to remember this design? Because for the resource server, like for any other authentication approach, you need to change the authentication provider if you want to customize the way the authentication works.

In the case of a resource server, Spring Security allows you to plug into the configuration a component named authentication manager resolver (figure 3.8). This component allows the app execution to decide which authentication manager to call. This way, you can delegate the authentication to any custom authentication manager which further can use a custom authentication provider.

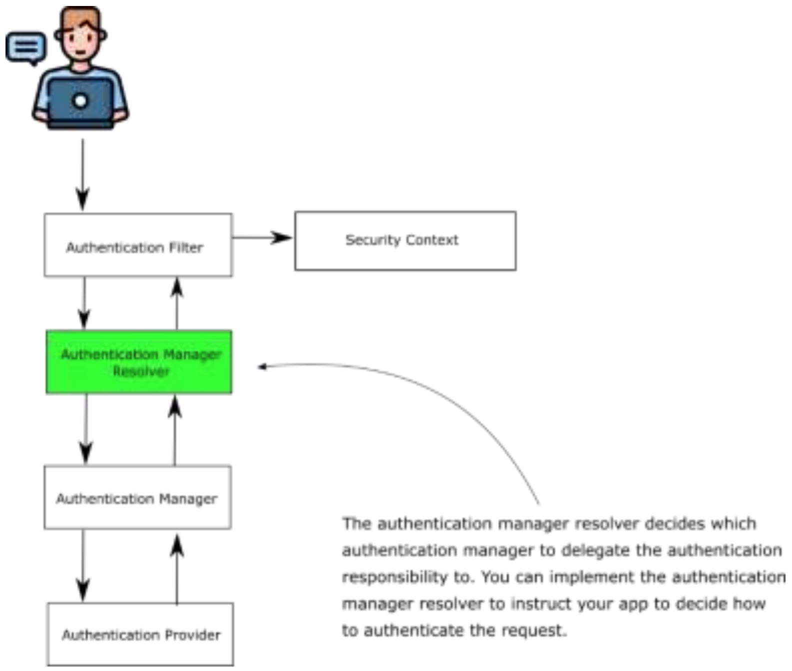

Authentication Filter
Security Context
Authentication Manager Resolver
Authentication Manager
Authentication Provider
The authentication manager resolver decides which authentication manager to delegate the authentication responsibility to. You can implement the authentication manager resolver to instruct your app to decide how to authenticate the request.

Figure 3.8 Implementing an authentication manager resolver, you tell your app which authentication manager to delegate the authentication responsibility to.

If you want your app to use multiple authorization servers all using JWT tokens, Spring Security even provides an out-of-the-box authentication manager resolver implementation (figure 3.9). For such a case, you only need to plug in the `JwtIssuerAuthenticationManagerResolver` custom implementation that Spring Security provides.

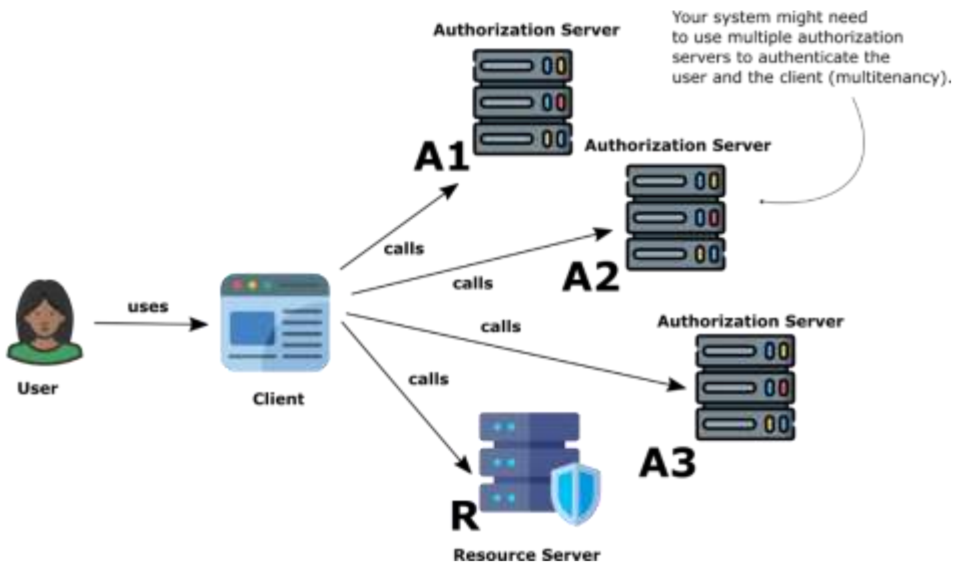

Authorization Server
Your system might need to use multiple authorization servers to authenticate the user and the client (multitenancy).
A1
Authorization Server
A2
calls
calls
calls
calls
uses
User
Client
Authorization Server
A3
R
Resource Server

Figure 3.9 Your system might need to use multiple authorization servers to authenticate the users and clients.

Listing 3.17 shows you how to use the `authenticationManagerResolver()` method when configuring the authentication. In this example, you observe that I only had to create an instance of the `JwtIssuerAuthenticationResolver` class, for which I have provided all the issues addresses of the authorization servers.

> **NOTE** Remember never to write URLs (or any similar configurable details) directly in code. We use this approach only with examples to simplify the code and allow you to focus on what's essential for you to learn. Anything customizable should always be written in configuration files or environment variables.

Listing 3.17 Working with two authorization servers that use JWT access tokens

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {


    http.oauth2ResourceServer(
      j -> j.authenticationManagerResolver(
                authenticationManagerResolver())
    );

    http.authorizeHttpRequests(
        c -> c.anyRequest().authenticated()
      );

      return http.build();
    }

    @Bean
    public AuthenticationManagerResolver<HttpServletRequest>
      [CA]authenticationManagerResolver() {

      var a = new JwtIssuerAuthenticationManagerResolver(
          "http://localhost:7070",
          "http://localhost:8080");

      return a;
    }
  }
```

With a configuration as the one presented in figure 3.17 your resource server works with two authorization servers running on ports 7070 and 8080.

However, sometimes things are more complex. Spring Security cannot provide any customization possible. In such a case where you need to customize the resource server's capabilities even further, you must implement your custom authorization manager resolver.

Let's consider the following scenario: you need your resource server to work with both JWT and opaque tokens with two different authorization servers. Say your resource server discriminates the requests based on the value of a "type" parameter. If the "type" parameter's value is "jwt" the resource server must authenticate the request with an authorization server using JWT access tokens, otherwise, it uses an authorization server with opaque access tokens.

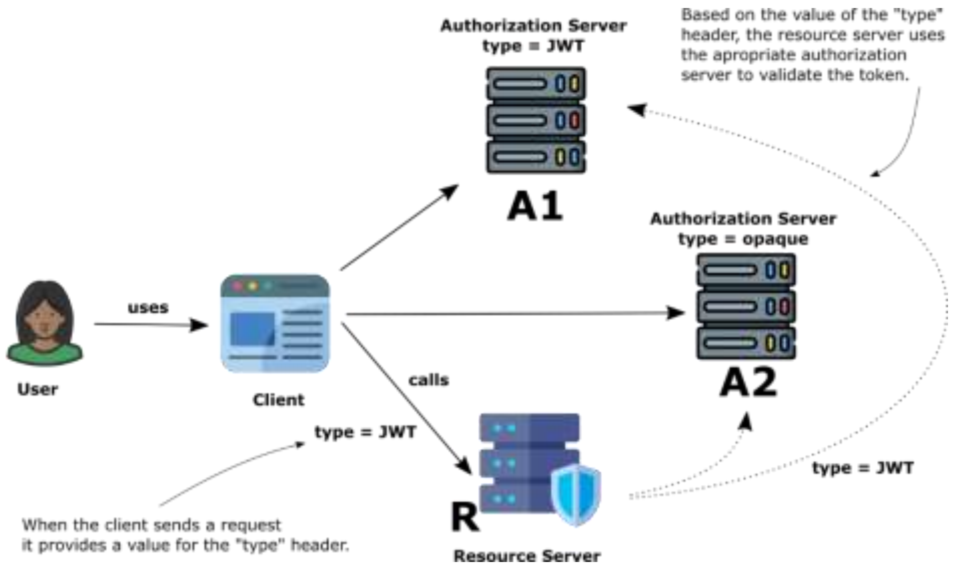

Authorization Server
type = JWT
Based on the value of the "type" header, the resource server uses the apropriate authorization server to validate the token.
A1
Authorization Server
type = opaque
uses
User
Client
calls
A2
type = JWT
type = JWT
R
Resource Server
When the client sends a request it provides a value for the "type" header.

Figure 3.10 Using two different authorization servers that manage two different token kinds. When the

client uses a certain value on a header of the HTTP request, the resource server know how to use a certain authorization server to validate the access tokens.

Listing 3.18 implements this scenario. The resource server uses a different authorization server based on the value of the "type" header in the HTTP request. To achieve this, the resource server uses a different authentication manager based on this header's value.

Listing 3.18 Using both JWT and opaque tokens

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public AuthenticationManagerResolver<HttpServletRequest>
    [CA]authenticationManagerResolver(
        JwtDecoder jwtDecoder,
        OpaqueTokenIntrospector opaqueTokenIntrospector
    ) {

    AuthenticationManager jwtAuth = new ProviderManager(    #A
      new JwtAuthenticationProvider(jwtDecoder)
    );

    AuthenticationManager opaqueAuth = new ProviderManager(     #B
      new OpaqueTokenAuthenticationProvider(opaqueTokenIntrospector)
    );

    return (request) -> {      #C
      if ("jwt".equals(request.getHeader("type"))) {
         return jwtAuth;
      } else {
         return opaqueAuth;
      }
    };
  }

  @Bean
  public JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder    #D
            .withJwkSetUri("http://localhost:7070/oauth2/jwks")
            .build();
  }

  @Bean
  public OpaqueTokenIntrospector opaqueTokenIntrospector() {     #E
    return new SpringOpaqueTokenIntrospector(
       "http://localhost:6060/oauth2/introspect",
       "client", "secret");
  }
}
```

#A Defining an authentication manager for the authorization server managing JWT access tokens.
#B Defining another authentication manager for the authorization server managing opaque tokens.

#C Defining the custom authentication manager resolver logic to pick one authentication manager or another based on the "type" header of the HTTP request.
#D Configuring the public key set URI for the authentication manager working with the authorization server that manages JWT access tokens.
#E Configuring the introspection URI and credentials for the authentication manager working with the authorization server that manages opaque tokens.

Listing 3.19 shows the rest of the configuration that configures the custom authorization manager resolver using the customizer parameter of the `authenticationManagerResolver()` method.

Listing 3.19 Configuring the AuthenticationManagerResolver

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {

    http.oauth2ResourceServer(
      j -> j.authenticationManagerResolver(     #A
                authenticationManagerResolver(
                  jwtDecoder(),
                  opaqueTokenIntrospector()
                ))
    );

    http.authorizeHttpRequests(
      c -> c.anyRequest().authenticated()
    );

    return http.build();
  }

  // Omitted code

}
```

#A Configuring the custom authentication manager resolver.

Even in this example, we used authentication provider implementations that Spring Security provides: `JwtAuthenticationProvider` and `OpaqueTokenAuthenticationProvider`. In this case, `JwtAuthenticationProvider` implements the authentication logic for working with a standard authorization server using JWT access tokens. `OpaqueTokenAuthenticationProvider` implements the authentication logic working with opaque tokens. But you could even have more complex cases in real-world apps.

If you need to implement something very custom, like integrating with a system that doesn't follow any standard, then you can even implement your custom authentication provider.

3.5 Summary

- Spring Security offers support for implementing OAuth 2/OpenID Connect resource servers. To configure authentication as an OAuth 2/OpenID Connect resource server, you use the `oauth2ResourceServer()` method of the `HttpSecurity` object.

- If you wish to use JWT tokens, you need to apply the configuration using the `jwt()` method of the `oauth2ResourceServer()` customizer parameter.
- You can use introspection, either because your system uses opaque tokens or because you want to be able to revoke JWT tokens at the authorization server side. In such a case, you must configure authentication using the `opaqueToken()` method of the `oauth2ResourceServer()` customizer parameter.
- When using JWT tokens, you must set up the public key set URI. The public key set URI is a URI exposed by the authorization server. The resource server calls this URI to get the public parts for the key pairs configured on the authorization server side. The authorization server uses the private parts to sign the access tokens, while the resource server needs the public parts to validate the tokens.
- When using introspection, you need to configure the introspection URI. The resource server sends requests to the introspection URI to "ask" the authorization server whether a token is valid and for more details about it. When calling the introspection URI, the resource server acts as a client for the authorization server, so it needs its own client credentials to authenticate.
- Spring Security offers you the possibility to customize the authentication logic using an authentication manager resolver component. You define and configure such a custom component when you have to implement a more specific case, such as multitenancy or adapting your app to a non-standard implementation.

4 Implementing an OAuth 2 client

This chapter covers

- Implementing OAuth 2 login
- Implementing a Spring Security OAuth 2 client
- Using the client credentials grant type

Often we need to implement communication between backend applications, especially with backend apps composed of multiple services. In such cases, when systems have authentication and authorization built over OAuth 2, it's recommended that you authenticate calls between apps using the same approach. While in some cases, developers use HTTP Basic and API Key authentication methods for simplicity, to keep the system consistent and more secure, using the OAuth 2 client credentials grant type is the preferred option for such cases.

Remember the OAuth 2 actors (figure 4.1)? We discussed the authorization server in chapter 2 and the resource server in chapter 3. This chapter is dedicated to the client. We'll discuss how to use Spring Security to implement an OAuth 2 client and when and how a backend app becomes a client in an OAuth 2 system.

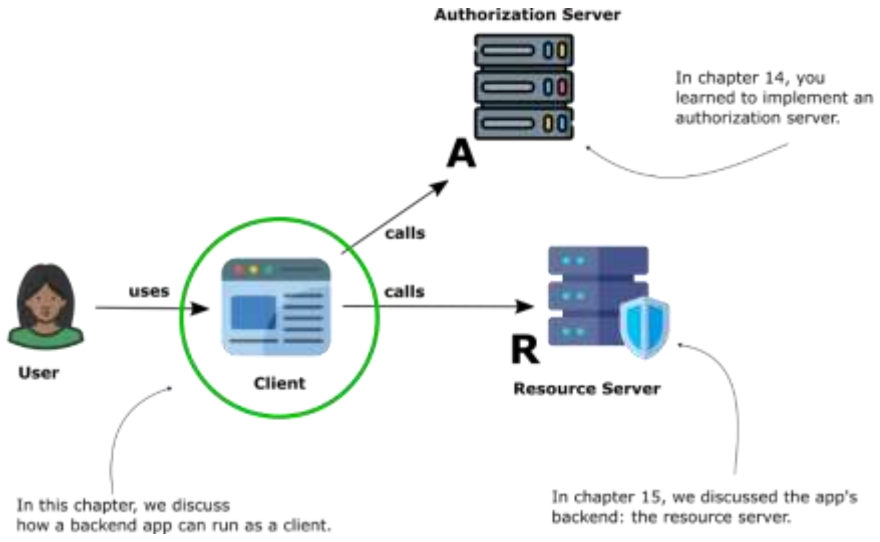

Authorization Server
A
In chapter 14, you learned to implement an authorization server.
calls
uses
calls
User
Client
R
Resource Server
In this chapter, we discuss how a backend app can run as a client.
In chapter 15, we discussed the app's backend: the resource server.

Figure 4.1 The OAuth2 actors. In this chapter, we discuss the client and how a backend app can act as a client in a system with authentication and authorization designed as OAuth 2.

OK! Maybe figure 4.1 is not entirely complete about what we will discuss. We’ll begin by discussing login for the user, but we’ll also focus on how to make a backend app a client for another backend app. Backend apps designed with Spring Security may become clients as well. Figure 4.2 shows the other case that we discuss in this chapter. In the current chapter, we solve the problem of implementing the communication between two backend apps, making one of them an actual OAuth 2 client. In such a case, we need to use Spring Security to build an OAuth 2 client.

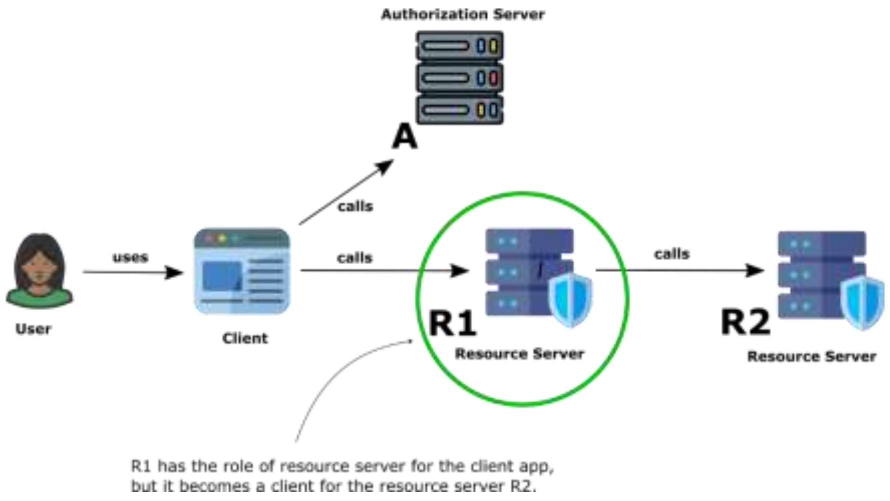

Authorization Server
A
calls
uses
calls
calls
User
Client
R1
Resource Server
R2
Resource Server
R1 has the role of resource server for the client app,
but it becomes a client for the resource server R2.

Figure 4.2 A backend app may become a client for another backend app. We discuss this case in the current chapter.

In section 4.1, we discuss how to easily implement an OAuth 2 login for a Spring MVC web app using Spring Security. We'll use an external authorization server provider such as Google and GitHub. You'll learn how to implement a login for your app where users can authenticate using their Google or GitHub credentials. Using the same approach, you can implement such a login with a custom (self-owned) authorization server.

In section 4.2, we leverage a client's custom implementation through a service and discuss using the client credentials grant type.

4.1 Implementing OAuth 2 login

In this section, we discuss implementing an OAuth 2 login for your Spring web app. With Spring Boot, it's a piece of cake to configure the authentication for standard cases (cases where the authorization server correctly fulfills the OAuth 2 and OpenID Connect specifications). We'll begin with a standard case (which you can use with most well-known providers such as Google, GitHub, Facebook, Okta).

Then, I'll show you what's behind the scenes of the self-provided configuration so you can also cover custom cases. At the end of this section, you'll be able to implement login for your Spring web app with any OAuth 2 provider and even allow your users to choose between various providers when authenticating.

4.1.1 Implementing authentication with a common provider

In this section, we'll implement the simplest login case, allowing our app's users to log in using only one provider. For this demonstration, I choose Google as our user's authentication provider.

We begin by adding a few resources to our project to implement a simple Spring web app with the mentioned login capabilities. The next code snippet shows you the dependencies for the demo app. You'll recognize a new dependency we haven't used in previous chapters – *the OAuth 2 client dependency*.

Listing 4.1 Needed dependencies for our demonstration

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-oauth2-client</artifactId>   #A
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

#A The only new dependency you observe is the OAuth 2 client dependency. We need this dependency for all the OAuth 2 client capabilities we configure in the project.

The following code snippet shows the simple controller of our demo web app, which only has a home page. If you need a refresher on building web apps with Spring Boot, chapters 7 and 8 of *Spring Start Here* (Manning, 2020), another book I wrote, should help you remember quickly these skills.

```
@Controller
public class HomeController {

  @GetMapping("/")
  public String home() {
    return "index.html";
  }
}
```

The next code snippet shows the small demo HTML page we expect to access once the authentication ends successfully.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <h1>Home</h1>
</body>
</html>
```

Listing 4.2 presents the configuration for OAuth 2 login as an authentication method for the web app. Configuring the app in such a way will automatically follow the authorization code grant type redirecting the user to log in on a specific authorization server and redirecting it back once the authentication succeeds.

Listing 4.2 Configuring the OAuth 2 login

```
@Configuration
public class SecurityConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {

    http.oauth2Login(Customizer.withDefaults());    #A

    http.authorizeHttpRequests(
       c -> c.anyRequest().authenticated());

    return http.build();
  }
}
```

#A To configure authentication as OAuth 2 login, we use the oauth2Login() method.

I bet you are wondering now: Shouldn't it still have to fill in all those details such as the authorization URL, token URL, client ID, client secret, and so on? Yes, all these details are needed. Fortunately, Spring Security helps you again. If your app uses one of the providers Spring Security considers to be well-known, most of these details are prefilled. You only need to configure the app's client credentials. Spring Security considers well-known the following providers:

- Google
- GitHub
- Okta
- Facebook

Spring Security preconfigures the details for these in the `CommonOAuth2Provider` class. So if you use any of these four, you only need to configure the client credentials in your application properties, which works. The following code snippet shows the two properties you need to configure the client ID and the client secret when using Google (I truncated my credentials' values).

```
spring.security.oauth2.client.registration.google.client-id=790…
spring.security.oauth2.client.registration.google.client-secret=GOC…
```

I implied here that you have registered your app in the Google developers console: that's where you get your app's unique set of credentials from. If you haven't done this before and you want to configure authentication using Google for your app, you can find Google's detailed documentation on how to register your OAuth 2 app with Google here: https://developers.google.com/identity/protocols/oauth2

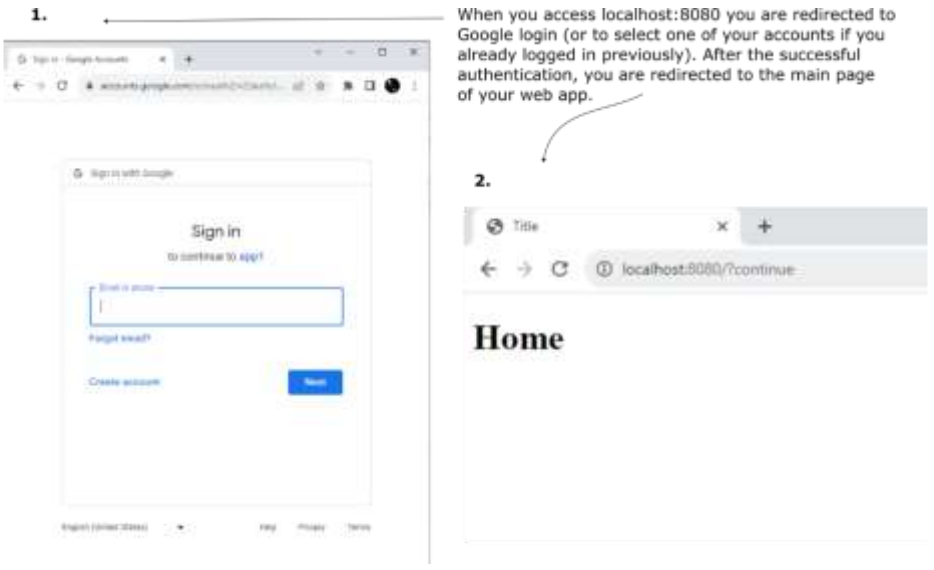

1.
When you access localhost:8080 you are redirected to Google login (or to select one of your accounts if you already logged in previously). After the successful authentication, you are redirected to the main page of your web app.
2.
Sign in
Next
Create account
Title
localhost:8080/?continue
Home

Figure 4.3 When accessing the app in a web browser, the browser redirects you to the Google login page. If you correctly authenticate with Google, you are redirected back to your app's main page.

4.1.2 Giving the user more possibilities

I'm sure you've surfed the internet enough by now to observe that many apps offer more than one possibility for the user to log in. Sometimes you can choose even among four or five providers to log into an app. This approach is advantageous since not all of us already have an account with one social network or another, for example. Some have a Facebook account, but others prefer to use LinkedIn. Some developers prefer to log in using their GitHub account, but others use their Gmail address.

With Spring Security, you can make this work straightforwardly, even with using multiple providers. Say I want to enable my app's users to log in with either Google or GitHub. I only need to configure the credentials for both providers similarly. The following snippet shows the properties we need to add to the application.properties file to add GitHub as an authentication method. Remember that you must keep those we already configured for Google in section 4.1.1.

```
spring.security.oauth2.client.registration.github.client-id=03…
spring.security.oauth2.client.registration.github.client-secret=c5d…
```

Similarly to any other provider, you have to register your app first and to the client ID and secret that you configure in the application.properties file. The approach for registering the app differs from one provider to another. For GitHub, you find the documentation that tells you how to register an app at the following link: https://docs.github.com/en/apps/creating-github-apps/about-creating-github-apps/about-creating-github-apps

Before asking you to authenticate, the app gives you two login options - the ones we previously configured (figure 4.4). You must choose either Google or GitHub to log in. After you choose your preferred provider, the app redirects you to that provider's specific authentication page.

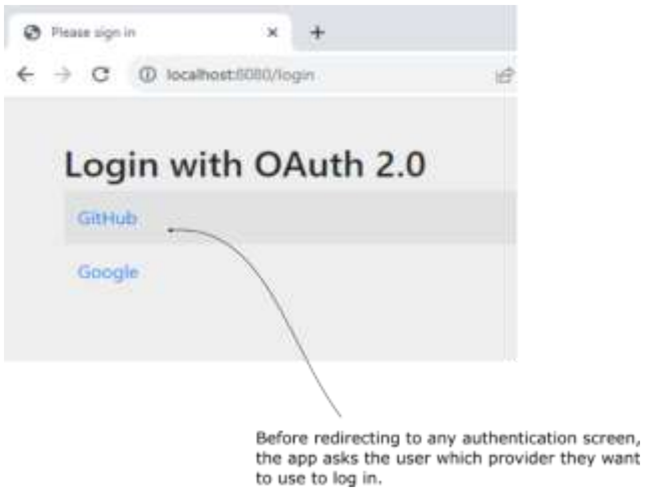

Please sign in
localhost:8080/login
Login with OAuth 2.0
GitHub
Google
Before redirecting to any authentication screen, the app asks the user which provider they want to use to log in.

Figure 4.4 The app allows the user to choose between using GitHub and Google when authenticating in the app.

4.1.3 Using a custom authorization server

Spring Security defines a list of four common providers, as discussed in sections 4.1.1 and 4.1.2. But what if you want to use a provider not on the common providers list? You have many other alternatives, such as LinkedIn, Twitter, Yahoo, and many others. You might want to use a custom authorization server you built, as you learned in chapter 2.

You can configure OAuth 2 login with any provider, including a custom one you built. In this section, we're going to use an authorization server we built in chapter 2 to show the configuration of a custom OAuth 2 login.

We only need to ensure that our client configuration matches what we want to implement in this chapter. Listing 4.2 shows the registered client configured in our authorization server. The most important thing here is to make sure that the redirect URI matches the one we expect for our application for which we'll implement the login: http://localhost:8080/login/oauth2/code/my_authorization_server

Figure 4.5 analyzes the anatomy of the redirect URI. Observe that the standard redirect URI uses the /login/oauth2/code path followed by the name of the authorization server. In this example, the name I gave to the authorization server is "my_authorization_server."

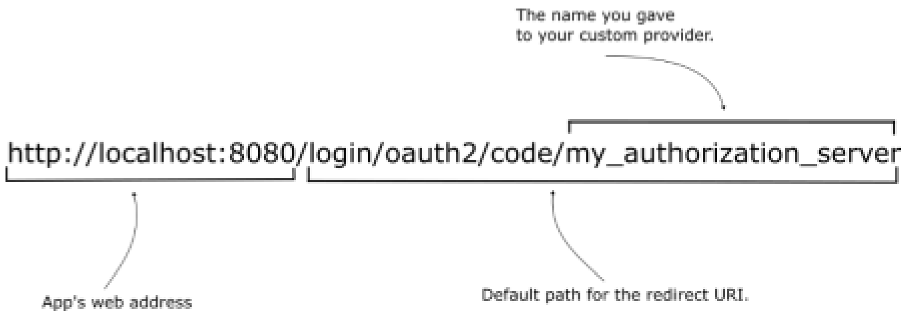

The name you gave to your custom provider.
http://localhost:8080/login/oauth2/code/my_authorization_server
App's web address
Default path for the redirect URI.

Figure 4.5 The standard redirect URI format. The last part of its path is the provider's name.

Listing 4.2 shows the configuration part from the authorization server, which registers the client details. You need these details when later in this section; we'll configure them at the app side as well.

Listing 4.2 The client details registered on the authorization server side

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
  var registeredClient = RegisteredClient
    .withId(UUID.randomUUID().toString())
    .clientId("client")
    .clientSecret("secret")
    .clientAuthenticationMethod(
       ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
    .authorizationGrantType(
       AuthorizationGrantType.AUTHORIZATION_CODE)
    .redirectUri(
      "http://localhost:8080/login/oauth2/code/my_authorization_server")
    .scope(OidcScopes.OPENID)
    .build();

    return new InMemoryRegisteredClientRepository(registeredClient);
  }
```

Remember, you can't start two apps using the same port number on the same system. Since the web app uses port 8080, we must change the authorization server's port to another one. As presented in the following code snippet, I chose 7070 for this example and configured it in the application.properties file.

```
server.port=7070
```

We can now move to the web app's configuration. Since we used a common provider in our examples from sections 4.1.1 and 4.1.2, we didn't have to define it. Spring Security already knows all the details it needs about common providers. But we need to configure a few things to use a different provider. Spring Security needs to know the following (as we discussed in chapters 1 and 2):

- The authorization endpoint for the provider to know where to redirect the user during the authorization code flow.

- The token endpoint that the app has to call to get an access token.
- The key set endpoint the app needs to validate the access tokens.

The good news is that if your provider (authorization server) correctly fulfills the OpenID Connect protocol, you only need to configure the issuer URI. The app then uses the issuer URI to find all details it needs, such as the authorization URI, token URI, and the key set URI. If the authorization server doesn't fulfill the OpenID Connect protocol, you'll have to configure these three details in the application.properties file explicitly.

Since the authorization servers we built in chapter 2 correctly implement the OpenID Connect protocol, we can rely on the issuer URI. The next code snippet shows you how to configure the issuer URI. Observe I gave a name to the provider. For this example, I chose to identify it with the name "my_authorization_server," but you can choose any name to identify your provider.

```
spring.security.oauth2.client.provider.my_authorization_server.issuer-
uri=http://127.0.0.1:7070
```

NOTE We run both apps, the authorization server and the web app we use on the local system. Running these apps on the same system and accessing them from the browser may cause issues with the cookies the browser uses to store the user's session. For this reason, I recommend you use the IP address "127.0.0.1" to refer to one app and the DNS name "localhost" to refer to the other. Even if the two are identical from a networking point of view and they refer to the same system (the local system) they will be considered different by the browser, which will, this way, be able to manage the sessions correctly. In this example, I use "127.0.0.1" to refer to the authorization server and "localhost" for the web app.

Listing 4.3 shows the client registration configuration. Apart from declaring who the provider is, the client registration is also a bit longer than the one we wrote in sections 4.1.1 and 4.1.2, where we used common providers. Aside from client ID and client secret, you also need to fill in:

- *The provider name* is a name you give to the provider you want to use in case it's not common.
- *The client authentication meeting* – the app's authentication method to call the provider's secured endpoints (usually HTTP Basic).
- *The redirect URI* – the URI the app expects the provider to redirect the user to after correct authentication. This URI has to match with one of those registered on the authorization server side (see listing 4.2).
- *The web app's requested scope* – The scope that the web app requests can only be one of those registered at the authorization server side (see listing 4.2).

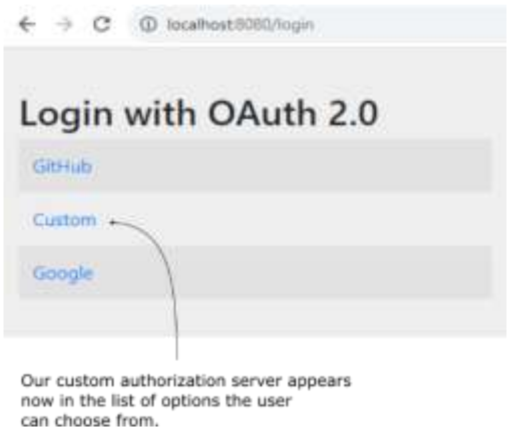
Listing 4.3 The client registration configuration

```
spring.security.oauth2.client.registration.my_authorization_server
```

```
.client-id=client     #A
spring.security.oauth2.client.registration.my_authorization_server.client-
name=Custom    #B
spring.security.oauth2.client.registration.my_authorization_server.client-
secret=secret    #C
spring.security.oauth2.client.registration.my_authorization_server.provider
=my_authorization_server    #D
spring.security.oauth2.client.registration.my_authorization_server.client-
authentication-method=client_secret_basic    #E
spring.security.oauth2.client.registration.my_authorization_server.redirect
-uri=http://localhost:8080/login/oauth2/code/my_authorization_server  #F
spring.security.oauth2.client.registration.my_authorization_server.scope[0]
=openid    #G
```

#A The client ID registered at the authorization server side.
#B The display name of the client.
#C The client secret registered at the authorization server side.
#D The name of the custom provider.
#E The app's authentication method to call provider's protected endpoints.
#F The URI the provider redirects the user to after successful authentication.
#G The scope the app requests.

You can start the authorization server and the web application. Remember, you must first start the authorization server. When the web app starts, it will call the issue URI to get the rest of the detail it needs. Once you start both apps, access the web app in the browser using its address http://localhost:8080. Figure 4.6 shows you that the custom provider now appears in the list and can be chosen by the user to authenticate.


localhost:8080/login

Login with OAuth 2.0

GitHub

Custom

Google

Our custom authorization server appears now in the list of options the user can choose from.

Figure 4.6 The custom authorization server now appears in the providers list that the user can select from to authenticate.

4.1.4 *Adding flexibility to your configurations*

Often we need to have more flexibility than the properties files offer us. Sometimes we need to be able to dynamically change the credentials without redeploying the app. In other cases, we want to turn specific providers on or off or even offer access to these based on given logic.

For such cases, adding the credentials in the properties file and allowing Spring Boot to do the magic for us doesn't work anymore.

However, if you know what happens behind the scenes, you can customize the provider's details as you wish. The only two types you must remember are

- `ClientRegistration` – you use this object to define the details the client needs to use the authorization server (credentials, redirect URI, authorization URI, and so on).
- `ClientRegistrationRepository` – you implement this contract to define the logic that retrieves the client registrations. You, for example, can implement a client registration repository to tell your app to get the client registrations from a database or a custom vault.

For this example, I keep things simple. I'll continue using the application.properties file, but I'll use different names for the properties to prove that it's no longer Spring Boot configuring things for us. However, even if straightforward, this example shows the same approach you'd use if you wanted to store the details in a database or grab them by calling a given endpoint. In any such case, you must appropriately implement the `ClientRegistrationRepository` contract.

You define the `ClientRegistrationRepository` component as a Spring bean. The app will use your implementation to get the client registration details. Listing 4.4 shows an example where I used an in-memory implementation. In this example, I do three things:

1. Inject the credentials values from the properties file,
2. create a `ClientRegistration` object with all the needed details, and then
3. configure it in an in-memory `ClientRegistrationRepository` implementation.

Listing 4.4 Implementing custom logic

```
@Configuration
public class SecurityConfig {

  @Value("${client-id}")
  private String clientId;     #A

  @Value("${client-secret}")
  private String clientSecret;    #A

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {

    http.oauth2Login(Customizer.withDefaults());

    http.authorizeHttpRequests(
      c -> c.anyRequest().authenticated()
    );

    return http.build();
  }
```

```
    @Bean
    public ClientRegistrationRepository clientRegistrationRepository() {
      return new InMemoryClientRegistrationRepository(
        this.googleClientRegistration());    #C
    }

    private ClientRegistration googleClientRegistration() {
      return CommonOAuth2Provider.GOOGLE.getBuilder("google")    #B
                  .clientId(clientId)
                  .clientSecret(clientSecret)
                  .build();
    }

  }
```

#A Injecting credentials values from the properties file.
#B Creating the client registration based on the template of the common provider Google.
#C Providing an in-memory repository implementation that contains the client registration details.

4.1.5 Managing authorization for OAuth 2 login

In this section, we discuss using the authentication details. In most cases, your app needs to know who logged in. This requirement is either for displaying things differently or applying various authorization restrictions. Fortunately, using `oauth2Login()` authentication method doesn’t differ in this regard to any other authentication method.

Remember the Spring Security authentication design we discussed starting with chapter 2 (reproduced as figure 4.7)? Successful authentication always ends with the app adding the authentication details to the security context. Using `oauth2Login()` makes no exception.

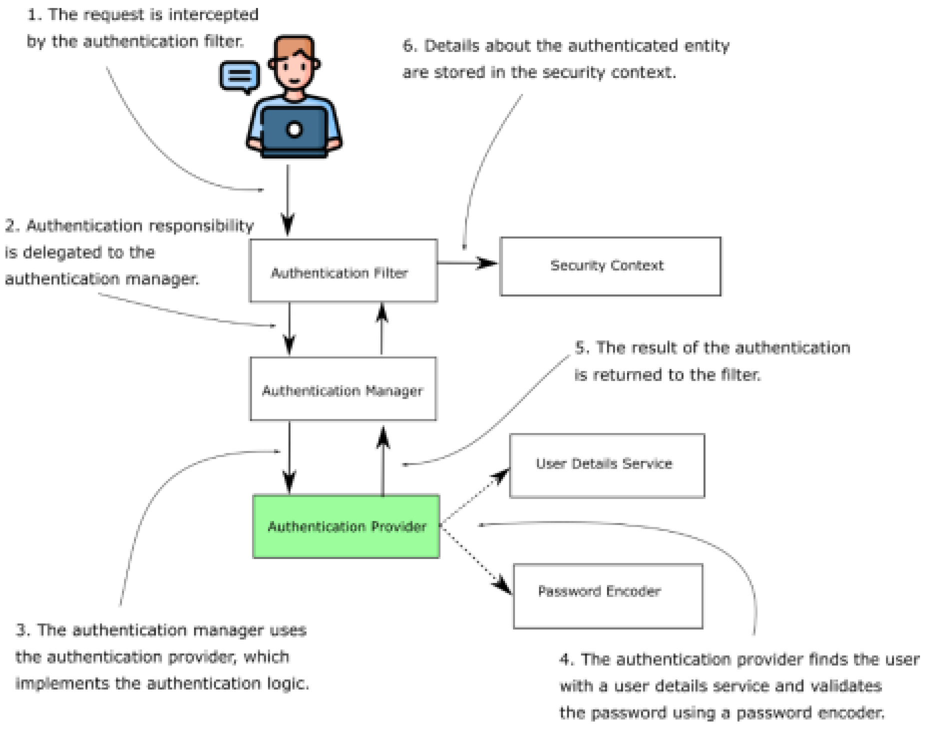

1. The request is intercepted by the authentication filter.
6. Details about the authenticated entity are stored in the security context.
2. Authentication responsibility is delegated to the authentication manager.
Authentication Filter
Security Context
5. The result of the authentication is returned to the filter.
Authentication Manager
User Details Service
Authentication Provider
Password Encoder
3. The authentication manager uses the authentication provider, which implements the authentication logic.
4. The authentication provider finds the user with a user details service and validates the password using a password encoder.

Figure 4.7 Authentication flow in Spring Security. Successful authentication ends with the app adding the details of the authenticated principal to the security context.

Knowing that the authentication details are in the security context, you can use them exactly the same way as for any other previously discussed authentication method: `httpBasic()`, `formLogin()`, or `oauth2ResourceServer()`:

- You can inject the `Authentication` object as a method parameter (figure 4.5)
- You can get it from the security context anywhere in the app (`SecurityContextHolder.getContext().getAuthentication()`)
- You can use pre/post annotations.

You can use the `Authentication` contract to get standard user details such as the user name and the authorities. If you need custom details, you can use the contract's implementation directly, as presented in listing 4.5. For OAuth 2, the class `OAuth2AuthenticationPrincipal` defines the contract implementation. Remember, however, that for maintainability purposes, I recommend you use the `Authentication` contract everywhere possible and rely on the implementation only if you have no other choice (for example, if you need to get a detail you can't get already using the contract reference.)

Listing 4.5 Obtaining the authentication details

```
@Controller
public class HomeController {

  @GetMapping("/")
  public String home(OAuth2AuthenticationToken authentication) {     #A
    // do something with the authentication
    return "index.html";
  }
}
```

#A Injecting authentication details in the method's parameter.

4.2 Implementing an OAuth 2 client

In this section, we discuss implementing a service as an OAuth 2 client. Often in service-oriented systems, apps communicate with one another. In such cases, the app sending the request to another app becomes a client of that particular app. In most cases, if we decide to implement authentication for the requests over OAuth 2, the app uses the client credentials grant type to obtain an access token.

The client credentials grant type doesn't imply a user. For this reason, you won't need a redirect URI and an authorization URI. The client credentials are enough to allow a client to authenticate and obtain an access token by sending a request to the token URI. Figure 4.8 reminds you of the client credentials grant type we discussed in chapter 1.

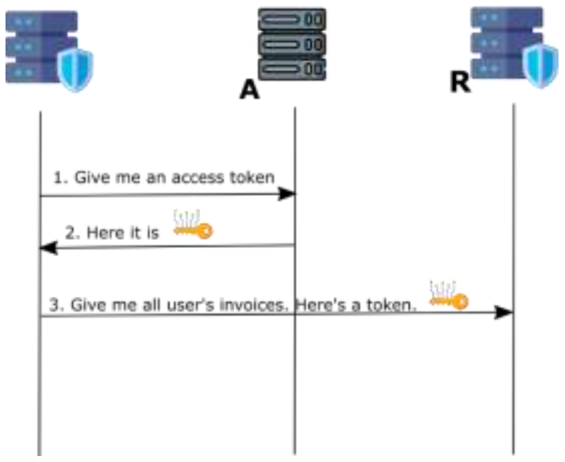

A
R
1. Give me an access token
2. Here it is
3. Give me all user's invoices. Here's a token.

Figure 4.8 The client credentials grant type. The client sends a request to the token endpoint using the client credentials to authenticate. After successful authentication, the client receives an access token it can use to access resources on the resource server side.

Let's build a simple example to show you everything you need to know about implementing OAuth 2 client capabilities with Spring Security. We'll build an app that uses the client credentials grant type to get an access token from an authorization server. This app will get an access token from an authorization server. To simplify the example, we'll only discuss retrieving the access token. It's irrelevant for our demonstration of how you create the request. As long as you've learned how to get an access token you can send the HTTP request either way, as any technology easily allows you to add a request header value (remember that you add the access token value to the *Authorization* request header prefixed with the string "Bearer").

So what we'll precisely do in this example is configure an app to retrieve an access token from an OAuth 2 authorization server using the client credentials grant type. To prove that we correctly retrieved the access token, we'll return it in the response body of a demo endpoint. Figure 4.9 visually shows what we want to build. The steps represented in the figure are:

1. The user (you) calls a demo endpoint we named /token using cURL (or an alternative tool like Postman).
2. The tool (cURL) that simulates an app sends the request to the application we build for this example.
3. Our application uses the client credentials grant type to retrieve an access token from an authorization server.
4. The app returns the access token's value to the client in the HTTP response body.
5. The user (you) finds the access token value in the HTTP response body.

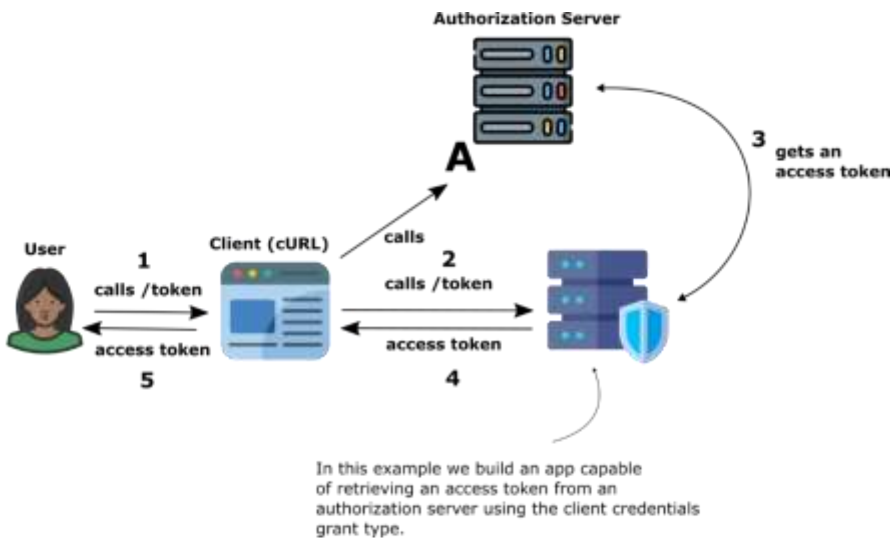

Authorization Server
A
3
gets an access token
User
1
calls /token
access token
5
Client (cURL)
calls
2
calls /token
access token
4
In this example we build an app capable of retrieving an access token from an authorization server using the client credentials grant type.

Figure 4.9 Our demonstration builds an app capable of retrieving an access token from an authorization server using the client credentials grant type. To prove that the app correctly retrieved the access token, the app sends the token value in response to a demo endpoint call. We named this demo endpoint /token.

Remember first to add to the authorization server a client registration that allows for using the client credentials grant type. You can change the one you configured previously in chapter 2 (as presented in listing 4.6) or add a second client registration that fulfills this requirement.

Listing 4.6 The client details registered on the authorization server side

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
  var registeredClient = RegisteredClient
    .withId(UUID.randomUUID().toString())
    .clientId("client")
    .clientSecret("secret")
    .clientAuthenticationMethod(
       ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
    .authorizationGrantType(
       AuthorizationGrantType.CLIENT_CREDENTIALS)     #A
    .scope(OidcScopes.OPENID)
    .build();

    return new InMemoryRegisteredClientRepository(registeredClient);
  }
```

#A Adding a client registration which allows the use of the client credentials grant type.

Similarly to other authentication methods, Spring Security offers a method of the `HttpSecurity` object to configure an app as OAuth 2 client. Call the `oauth2Client()` method presented in listing 4.7 to configure the app as an OAuth 2 client.

Listing 4.7 Configuring OAuth 2 client authentication

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {

    http.oauth2Client(Customizer.withDefaults());    #A

    http.authorizeHttpRequests(
       c -> c.anyRequest().permitAll()
    );

    return http.build();
  }

}
```

#A Using the oauth2Client() authentication method makes this app an OAuth 2 client.

The app also needs to know some details to send the authorization server the requests for access tokens. As you learned in section 4.1, we provide these details using a `ClientRegistrationRepository` component. You might find the code in listing 4.8 familiar since it resembles the code we wrote in listing 4.4.

However, since I don’t use a common provider, I had to specify more details, such as the scope, the token URI, and the authentication method. Observe that I configured client credentials as a grant type.

Listing 4.8 Configuring the client registration details for the client app

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public ClientRegistrationRepository clientRegistrationRepository() {
    ClientRegistration c1 =
      ClientRegistration.withRegistrationId("1")
        .clientId("client")
        .clientSecret("secret")
        .authorizationGrantType(AuthorizationGrantType.CLIENT_CREDENTIALS)
        .clientAuthenticationMethod(
            ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
        .tokenUri("http://localhost:7070/oauth2/token")
        .scope(OidcScopes.OPENID)
        .build();

    var repository =
        new InMemoryClientRegistrationRepository(c1);

    return repository;
  }

}
```

A client manager component makes the needed request for obtaining an access token. Figure 4.10 visually represents the relationship between the controller and the client manager (for our example).

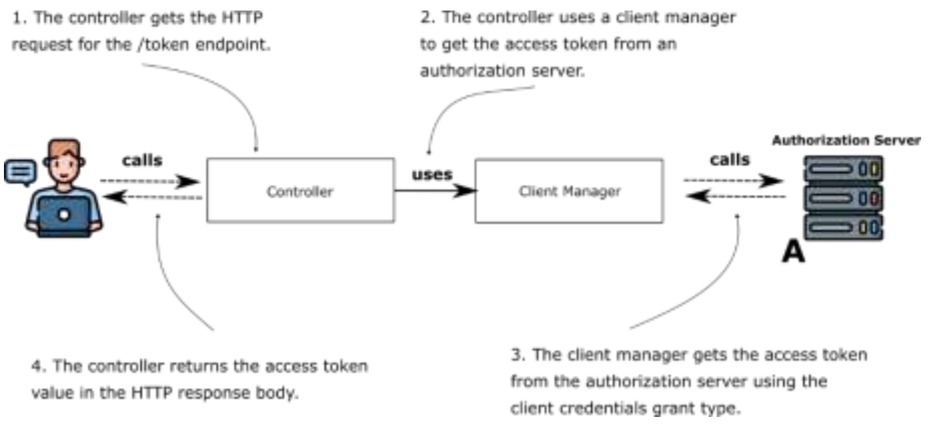

1. The controller gets the HTTP request for the /token endpoint.
2. The controller uses a client manager to get the access token from an authorization server.
calls
Controller
uses
Client Manager
calls
Authorization Server
A
4. The controller returns the access token value in the HTTP response body.
3. The client manager gets the access token from the authorization server using the client credentials grant type.

Figure 4.10 The controller uses a client manager to get the access token from an authorization server. The client manager is the Spring Security component responsible for connecting to the authorization server and

properly using the grant type to get the access token.

The class `OAuth2AuthorizedClientManager` defines a client manager. Listing 4.9 configures a client manager as a bean in the app's context.

Listing 4.9 Implementing an OAuth 2 client manager

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public OAuth2AuthorizedClientManager oAuth2AuthorizedClientManager(
    ClientRegistrationRepository clientRegistrationRepository,
    OAuth2AuthorizedClientRepository auth2AuthorizedClientRepository
  ) {

    var provider =    #A
        OAuth2AuthorizedClientProviderBuilder.builder()
            .clientCredentials()
            .build();

    var cm = new DefaultOAuth2AuthorizedClientManager(    #B
            clientRegistrationRepository,
            auth2AuthorizedClientRepository);

    cm.setAuthorizedClientProvider(provider);    #C

    return cm;
  }
}
```

#A Creating a provider object to specify the grant types intended to be used.
#B Creating a client manager instance that will handle the client request logic.
#C Setting the provider for the client manager.

You can now use the client manager wherever you need to get an access token. As shown in figure 4.10, I made the controller use the client manager directly to simplify this example and allow you to focus on the discussion about implementing an OAuth 2 client. Remember that a real-world app would probably be more complex. In a design that correctly segregates the object's responsibilities, the client manager would likely be used by a proxy object and not by a controller directly (figure 4.11).

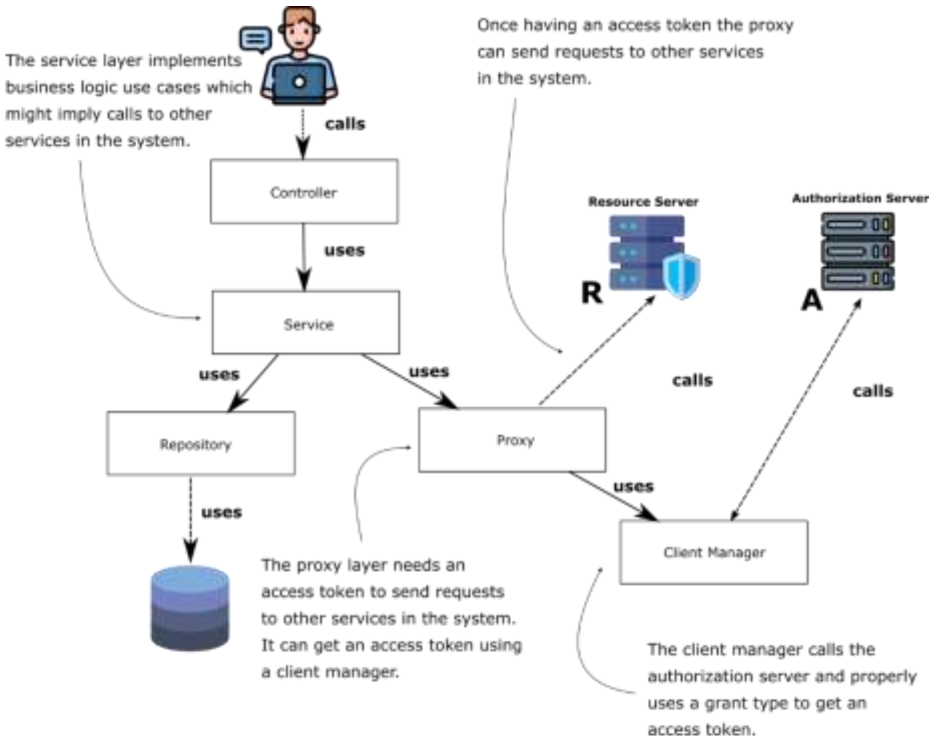

The service layer implements business logic use cases which might imply calls to other services in the system.
calls
Controller
uses
Service
uses
uses
Repository
uses
Proxy
uses
Client Manager
Once having an access token the proxy can send requests to other services in the system.
Resource Server
R
Authorization Server
A
calls
calls
The proxy layer needs an access token to send requests to other services in the system. It can get an access token using a client manager.
The client manager calls the authorization server and properly uses a grant type to get an access token.

Figure 4.11 A real-world app would have better-separated responsibilities. Unlike our example, a proxy layer uses the token it obtains with the help of a client manager to send requests to another app in the system.

Listing 4.10 shows you how to inject the client manager instance and demonstrates the retrieval of an access token using an endpoint. When calling the /token endpoint the app exposes, the response body should contain the value of the access token.

Listing 4.10 Using the OAuth 2 client manager to get a token

```
@RestController
public class DemoController {

  private final OAuth2AuthorizedClientManager clientManager;

  // Omitted constructor

  @GetMapping("/token")     #A
  public String token() {
    OAuth2AuthorizeRequest request = OAuth2AuthorizeRequest
        .withClientRegistrationId("1")
        .principal("client")
        .build();     #B
```

```
      var client =
         clientManager.authorize(request);      #C

      return client
        .getAccessToken().getTokenValue();    #D
    }
  }
```

#A Exposing a GET endpoint at the /token path.
#B Creating an authorization request instance.
#C Sending the request, the app returns the access token value.
#D The app returns the access token value in the response body.

Use the following cURL command to call the endpoint that the app exposes.

```
curl http://localhost:8080/token
```

The response body should contain the value of an access token, similar to what the next snippet presents.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Im
JpbGwiLCJpYXQiOjE1MTYyMzkwMjJ9.zjL2JXw0TVgNgTMUKmP0-PTPklULUVmV_5re50eZoHw
```

4.3 Summary

- When implementing a Spring web app, we often have to configure authentication capabilities. While we can quickly implement a login form with the `formLogin()` method, we can also allow users to authenticate using another system with a registered account.
- Allowing users to choose a different system to log in offers advantages for both the user and our app. The users don't need to remember supplementary credentials, and our app doesn't have to manage credentials for all their users.
- Spring Security considers GitHub, Google, Facebook, and Okta common providers. For the common providers, Spring Security already knows all the details to establish requests over the OAuth 2 framework, so you only need to configure the client credentials the provider offers to configure the login capability.
- You can configure your app to use providers other than the common ones, but you need to explicitly configure all the details the app needs to establish the grant type flows to get access tokens. The main details you need to configure are three URIs: the authorization URI, the token URI, and the key set URI.
- Once the user logs into your app, even if it is authenticated through an external system, the app gets details about them and stores them in the security context. This process follows the standard Spring Security authentication design. For this reason, you can configure authorization similarly to all the other authentication methods.
- Sometimes, a backend service becomes a client for another backend app. In such a case, an app that wants to call another app and use an OAuth 2 approach needs to get an access token to get authenticated by the latter. A service can use the client credentials grant type to get an access token.
- Spring Security provides an object called a client manager. This object implements the logic for executing a specific grant type and getting an access token. An app's proxy

layer that sends requests to another app and needs to authenticate the requests using access tokens would use a client manager to get the access token.