**Part 4. Securing, testing, and deploying microservice APIs**

As we learned in chapter 1, APIs are programmatic interfaces to our applications, and making our APIs public allows other organizations to build integrations with our own APIs. The growing offering of APIs as a means of delivering software products has given rise to the API economy. APIs open new opportunities for business growth, but they also represent a security risk. Lack of proper testing or wrongly implemented security protocols render our APIs vulnerable. Part 4 of this book will get you up and running on the major topics of API testing, security, and operations.

The modern standard for API authentication is OpenID Connect, and for API authorization it’s Open Authorization (OAuth) 2.1. Chapter 11 kicks off part 4 by introducing these standards. In my experience, this is one of the most misunderstood areas of API development, which leads to security vulnerabilities and breaches. Chapter 11 teaches you everything you need to know to implement a robust API authentication and authorization strategy for your APIs.

When you drive integrations using APIs, you need a reliable API testing and validation method. You must ensure that your API backend serves the interface defined in your API specification. How do we do that? As you’ll learn in chapter 12, a powerful approach to API testing is using contract-testing tools, such as Dredd and Schemathesis, and applying property-based testing. With these strategies, you can test and validate your code with confidence before releasing it to production.

Finally, what about deployments and operations? The final chapters of this book teach you how to Dockerize and deploy your microservice APIs using Kubernetes. You’ll learn to deploy and operate a Kubernetes cluster using AWS EKS, one of the most popular solutions for running Kubernetes in the cloud. After reading part 4, you’ll be ready to test, protect, and operate your microservice APIs at scale.

# 11 API authorization and authentication

This chapter covers

* Using Open Authorization to allow access to our APIs
* Using OpenID Connect to verify the identity of our API users
* What kinds of authorization flows exist, and which flow is more suitable for each authorization scenario
* Understanding JSON Web Tokens (JWT) and using Python’s PyJWT library to produce and validate them
* Adding authentication and authorization middleware to our APIs

In 2018, a weakness in the API authentication system of the US postal system ([https://usps.com](https://usps.com/)) allowed hackers to obtain data from 60 million users, including their email addresses, phone numbers, and other personal details.[**1**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/11.htm#pgfId-1111730) API security attacks like this have become more and more common, with an estimated growth of over 300% in the number of attacks in 2021.[**2**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/11.htm#pgfId-1111733) API vulnerabilities don’t only risk exposing sensitive data from your users; they can also put you out of business![**3**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/11.htm#pgfId-1111736) The good news is there are steps you can take to reduce the risk of an API breach. The first line of defense is a robust authentication and authorization system. In this chapter, you’ll learn to prevent unauthorized access to your APIs by using standard authentication and authorization protocols.

In my experience, API authentication and authorization are two of the most confusing topics for developers, and they’re also areas where implementation mistakes happen often. Before you implement the security layer of your API, I highly recommend you read this chapter to make sure you know what you’re doing and know how to do it correctly. I’ve done my best to provide a comprehensive summary of how API authentication and authorization work, and by the end of this chapter you should be able to add a robust authorization flow to your own APIs.

Authentication is the process of verifying the identity of a user, while authorization is the process of determining whether a user has access to certain resources or operations. The concepts and standards about authentication and authorization that you’ll learn in this chapter are applicable to all types of web APIs.

You’ll learn different authentication and authorization protocols and flows and how to validate authorization tokens. You’ll also learn to use Python’s PyJWT library to produce signed tokens and to validate them. We’ll walk through a practical example of adding authentication and authorization to the orders API. We’ve got a lot to cover, so let’s get started!

## 11.1 Setting up the environment for this chapter

Let’s set up the environment for this chapter. The code for this chapter is available under the directory called ch11 in the GitHub repository for this book. In chapter 7, we implemented a fully functional orders service, complete with a business layer, database, and API. This chapter picks up the orders service from where we left it in chapter 7. If you want to follow along with the changes in this chapter, copy over the code from chapter 7 into a new folder called ch11:

$ cp -r ch07 ch11

cd into ch11 and install the dependencies by running pipenv install. For this chapter, we need a few additional dependencies, so run the following command to install them:

$ pipenv install cryptography pyjwt

PyJWT is a Python library that allows us to work with JSON Web Tokens, while cryptography will allow us to verify the tokens’ signatures. (For a list of alternative JWT libraries in the Python ecosystem, check out [https://jwt.io/libraries?language=Python](https://jwt.io/libraries?language=Python.).)

Our environment is now ready, so let’s begin our quest through the wondrous world of user authentication and authorization. It’s a journey full of pitfalls, but a necessary one. Hold tight, and watch carefully as we go along!

## 11.2 Understanding authentication and authorization protocols

When it comes to API authentication, the two most important protocols you need to know are OAuth (Open Authorization) and OpenID Connect (OIDC). This section explains how each protocol works and how they fit within the authentication and authorization flows for our APIs.

### 11.2.1 Understanding Open Authorization

OAuth is a standard protocol for access delegation.[**4**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/11.htm#pgfId-1111761) As you can see in figure 11.1, OAuth allows a user to grant a third-party application access to protected resources they own in another website without having to share their credentials.

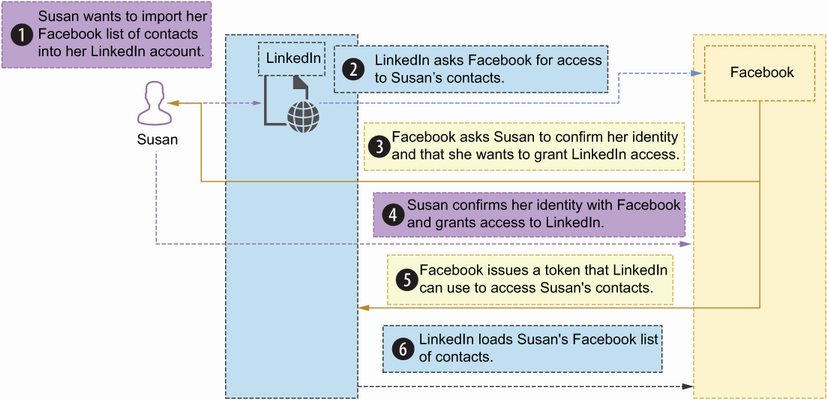


Figure 11.1 With OAuth, a user can grant a third-party application access to their information on another website.

**DEFINITION** *OAuth* is an open standard that allows users to grant access to third-party applications to their information on other websites. Typically, access is granted by issuing a token, which the third-party application uses to access the user’s information.

For example, let’s say Susan has a list of contacts in her Facebook account. One day, Susan signs into LinkedIn, and she wants to import her list of contacts from Facebook. To allow LinkedIn to import her Facebook contacts, Susan has to grant LinkedIn access to that resource. How can she grant LinkedIn access to her list of contacts? She could give LinkedIn her Facebook credentials to access her account. But that would be a major security risk. Instead, OAuth defines a protocol that allows Susan to tell Facebook that LinkedIn can access her list of contacts. With OAuth, Facebook issues a temporary token LinkedIn can use to import Susan’s contacts.

OAuth distinguishes various roles in the process of granting access to a resource:

* *Resource owner*—The user who’s granting access to the resource. In the previous example, Susan is the resource owner.
* *Resource server*—The server hosting the user’s protected resources. In the previous example, Facebook is the resource server.
* *Client*—The application or server requesting access to the user’s resources. In the previous example, LinkedIn is the client.
* *Authorization server*—The server that grants the client access to the resources. In the previous example, Facebook is the authorization server.

OAuth offers four different flows to grant authorization to a user depending on the access conditions. It’s important to know how each flow works and in which scenarios you can use it in. In my experience, OAuth flows are one of the biggest areas of confusion around authorization, and one of the biggest sources of security problems in modern websites. These are the OAuth flows:

* Authorization code flow
* PKCE flow
* Client credentials flow
* Refresh token flow

OAuth

OAuth flows are the strategies that a client application uses to authorize their access to an API. Best practices in OAuth change over time as we learn more about application vulnerabilities and we improve the protocol. Current best practices are described in IETF’s “OAuth 2.0 Security Best Current Practice” (<http://mng.bz/o58v>), written by T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett. If you read about OAuth 2.0, you may encounter references to two flows that we don’t describe in this chapter: the resource owner password flow and the implicit flow. Both are now deprecated since they expose serious vulnerabilities, and therefore you shouldn’t use them.

Another popular extension that we don’t discuss in this chapter is the device authorization grant (<http://mng.bz/5mZD>), which allows input-constrained devices such as smart TVs to obtain access tokens. The latest version of OAuth is 2.1, which is described in the IETF’s “The OAuth 2.1 Authorization Framework” (<http://mng.bz/69m6>).

Let’s delve into each flow to understand how they work and when we use them!

Authorization code flow

In the authorization code flow, the client server exchanges a secret with the authorization server to produce a signing URL. As you can see in figure 11.2, after the user signs in using this URL, the client server obtains a one-time code it can exchange for an access token. This flow uses a client secret, and therefore is only appropriate for applications in which the code is not publicly exposed, such as traditional web applications where the user interface is rendered in the backend. OAuth 2.1 recommends using the authorization code flow in combination with PKCE, which is described in the next section.

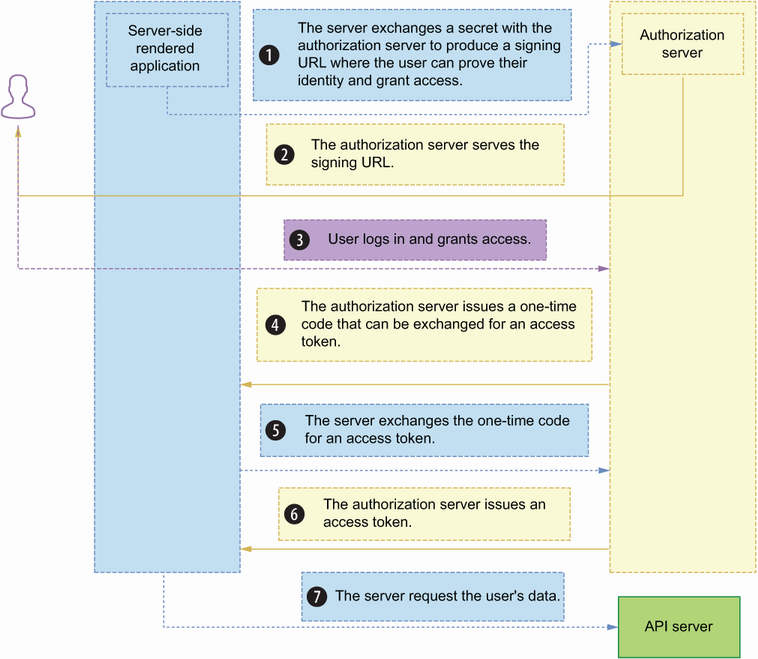


Figure 11.2 In the authorization code flow, the authorization server produces a signing URL, which the user can use to prove their identity and grant access to the third-party application.

Proof of Key for Code Exchange flow

The *Proof of Key for Code Exchange* (PKCE, pronounced “pixie”) is an extension of the authorization code flow designed to protect applications whose source code is publicly exposed, such as mobile applications and single-page applications (SPAs).[**5**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/11.htm#pgfId-1111807) Since the source is publicly exposed, the client cannot use a secret because it would also be publicly exposed.

As you can see in figure 11.3, in the PKCE flow, the client generates a secret called the *code verifier*, and it encodes it. The encoded code is called the *code challenge*. When sending an authorization request to the server, the client includes both the code verifier and the code challenge in the request. In return, the server produces an *authorization code*, which the client can exchange for an access token. To get the access token, the client must send both the authorization code and the code challenge.

Thanks to the code challenge, the PKCE flow also prevents authorization code injection attacks, in which a malicious user intercepts the authorization code and uses it to get hold of an access token. Due to the security benefits of this flow, PKCE is also recommended for server-side applications. We’ll see an example of this flow using an SPA in appendix C.

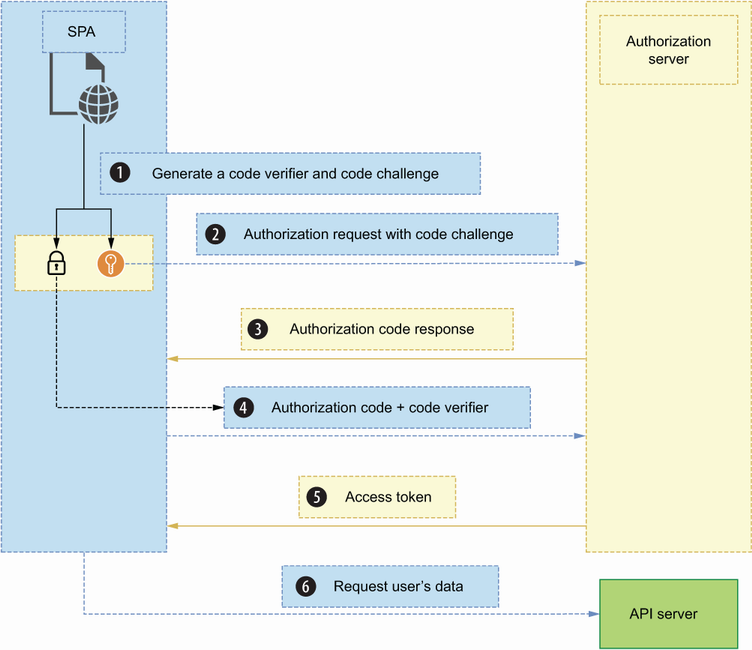


Figure 11.3 In the PKCE flow, an SPA served by the client requests access to the user’s data directly from the authorization server by exchanging a code verifier and a code challenge.

Client credentials flow

The client credentials flow is aimed for server-to-server communication, and as you can see in figure 11.4, it involves the exchange of a secret to obtain an access token. This flow is suitable for enabling communication between microservices over a secure network. We’ll see an example of this flow in appendix C.

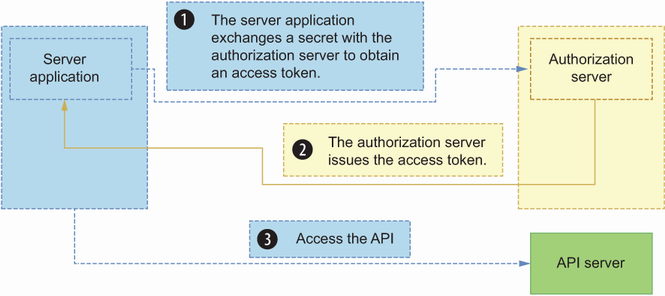


Figure 11.4 In the client credentials flow, a server application exchanges a secret with the authorization server to obtain an access token.

Refresh token flow

The refresh token flow allows clients to exchange a refresh token for a new access token. For security reasons, access tokens are valid for a limited period of time. However, API clients often need to be able to communicate with the API server after an access token has expired, and to obtain the new token they use the refresh token flow.

As you can see in figure 11.5, API clients typically receive both an access token and a refresh token when they successfully gain access to the API. Refresh tokens are usually valid for a limited period of time, and they’re valid for one-time use. Every time you refresh your access token, you’ll get a new refresh token.

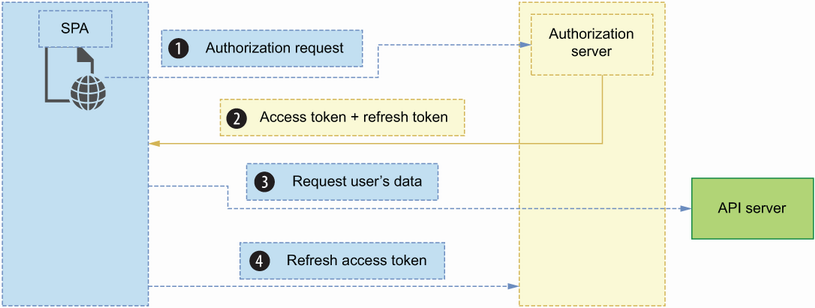


Figure 11.5 To allow API clients to use refresh tokens to continue communicating with the API server after the access token has expired, the authorization server issues a new refresh token every time the client requests a new access token.

Now that we understand how OAuth works, let’s turn our attention to OpenID Connect!

### 11.2.2 Understanding OpenID Connect

OpenID Connect (OIDC) is an open standard for identity verification that’s built on top of OAuth. As you can see in figure 11.6, OIDC allows users to authenticate to a website by using a third-party identity provider. If you’ve used your Facebook, Twitter, or your Google account to sign into other websites, you’re already familiar with OIDC. In this case, Facebook, Twitter, and Google are identity providers. You use them to bring your identity to a new website. OIDC is a convenient authentication system since it allows users to use the same identity across different websites without having to create and manage new usernames and passwords.

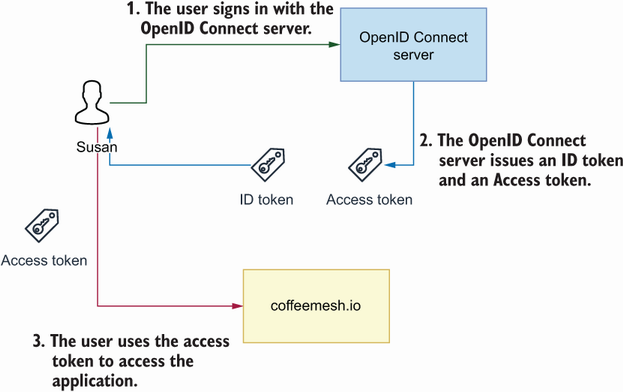


Figure 11.6 With OIDC, a user signs in with an OIDC server. The OIDC server issues an ID token and an access token, which the user can use to access an application.

**DEFINITION** *OpenID Connect* (OIDC) is an identity verification protocol that allows users to bring their identity from one website (the identity provider) to another. OIDC is built on top of OAuth, and we can use the same flows defined by OAuth to authenticate users.

Since OIDC is built on top of OAuth, we can use any of the authorization flows described in the previous section to authenticate and authorize users. As you can see in figure 11.6, when we authenticate using the OIDC protocol, we distinguish two types of tokens: ID tokens and access tokens. Both tokens come in the form of JSON Web Tokens, but they serve different purposes: *ID tokens* identify the user, and they contain information such as the user’s name, their email, and other personal details. You use ID tokens only to verify the user identity, and never to determine whether a user has access to an API. API access is validated with access tokens. *Access tokens* typically don’t contain user information but a set of claims about the access rights of the user.

**ID TOKENS VS. ACCESS TOKENS** A common security problem is the misuse of ID tokens and access tokens. ID tokens are tokens that carry the identity of the user. They must be used exclusively for verifying the user’s identity and not for validating access to an API. API access is validated through access tokens. Access tokens rarely contain a user’s identity details, and instead contain claims about the user’s right to access the API. A fundamental difference between ID tokens and access tokens is the audience: the ID token’s audience is the authorization server, while the access token’s audience is our API server.

Identity providers that offer OIDC integrations expose a /.well-known/openid-configuration endpoint (with a leading period!), also known as the *discovery endpoint*, which tells the API consumer how to authenticate and obtain their access tokens. For example, the OIDC’s well-known endpoint for Google Accounts is <https://accounts.google.com/.well-known/openid-configuration>. If you call this endpoint, you’ll obtain the following payload (the example is truncated with an ellipsis):

{

"issuer": "https://accounts.google.com",

"authorization\_endpoint": "https://accounts.google.com/o/oauth2/v2/auth",

"device\_authorization\_endpoint":

➥ "https://oauth2.googleapis.com/device/code",

"token\_endpoint": "https://oauth2.googleapis.com/token",

"userinfo\_endpoint": "https://openidconnect.googleapis.com/v1/userinfo",

"revocation\_endpoint": "https://oauth2.googleapis.com/revoke",

"jwks\_uri": "https://www.googleapis.com/oauth2/v3/certs",

"response\_types\_supported": [

"code",

"token",

"id\_token",

"code token",

"code id\_token",

"token id\_token",

"code token id\_token",

"none"

],

...

}

As you can see, the well-known endpoint tells us which URL we must use to obtain the authorization access token, which URL returns user information, or which URL we use to revoke an access token. There are other bits of information in this payload, such as available claims or the JSON Web Keys URI (JWKS). Typically, you use a library to handle these endpoints on your behalf, or you use an identity-as-a-service provider to take care of these integrations. If you want to learn more about OpenID Connect, I recommend Prabath Siriwardena’s *OpenID Connect in Action* (Manning, 2022).

Now that we know how OAuth and OpenID Connect work, it’s time get into the details of how authentication and authorization work. We’ll start by studying what JSON Web Tokens are in the next section.

## 11.3 Working with JSON Web Tokens

In OAuth and OpenID Connect, user access is verified by means of a token known as *JSON Web Token*, or JWT. This section explains what JSON Web Tokens are, how they’re structured, what kinds of claims they contain, and how to produce and validate them.

A JWT is a token that represents a JSON document. The JSON document contains claims, such as who issued the token, the audience of the token, or when the token expires. The JSON document is typically encoded as a Base64 string. JWTs are normally signed with a private secret or a cryptographic key.[**6**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/11.htm#pgfId-1111905) A typical JSON Web Token looks like this:

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29mZmVlbW

➥ VzaC5pby8iLCJzdWIiOiJlYzdiYmNjZi1jYTg5LTRhZjMtODJhYy1iNDFlNDgzMWE5NjIiL

➥ CJhdWQiOiJodHRwOi8vMTI3LjAuMC4xOjgwMDAvb3JkZXJzIiwiaWF0IjoxNjM4MjI4NDg2

➥ LjE1Otg4MSwiZXhwIjoxNjM4MzE0Odg2LjE1Otg4Mswic2NvcGUiOiJvcGVuaWQifQ.oblJ

➥ 5wV9GqrhIDzNSzcClrpEQTMK8hZGzn1S707tDtQE\_\_OCDsP9J2Wa70aBua6X81-

➥ zrvWBfzrcX--nSyT-

➥ A9uQxL5j3RHHycToqSVi87I9H6jgP4FEKH6ClwZfabVwzNIy52Zs7zRdcSI4WRz1OpHoCM-

➥ 2hNtZ67dMJQgBVIlrXcwKAeKQWP8SxSDgFbwnyRTZJt6zijRnCJQqV4KrK\_M4pv2UQYqf9t

➥ Qpj2uflTsVcZq6XsrFLAgqvAg-YsIarYw9d63rs4H\_I2aB3\_T\_1dGPY6ic2R8WDT1\_Axzi-

➥ crjoWq9A51SN-kMaTLhE\_v2MSBB3A0zrjbdC4ZvuszAqQ

If you look closely at the example, you’ll see the string contains two periods. The periods act as delimiters that separate each component of the JSON Web Token. As you can see in figure 11.7, a JSON Web Token document has three sections:

* *Header*—Identifies the type of token as well as the algorithm and the key that were used sign the token. We use this information to apply the right algorithm to verify the token’s signature.
* *Payload*—Contains the document’s set of claims. The JWT specification includes a list of reserved claims that identify the issuer of the token (the authorization server), the token’s audience or intended recipient (our API server), and its expiry date, among other details. In addition to JWT’s standard claims, a payload can also include custom claims. We use this information to determine whether the user has access to the API.
* *Signature*—A string representing the token’s signature.

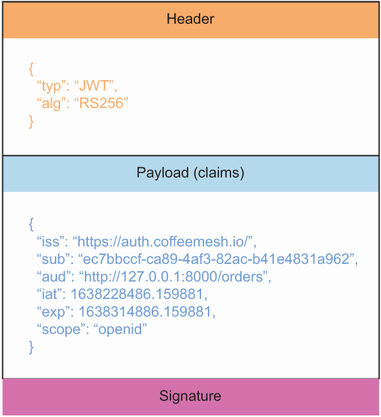


Figure 11.7 A JWT is composed of three parts: a header that contains information about the token itself, a payload with claims about the user’s access to the website, and a signature that proves the authenticity of the token.

Now that we understand what a JWT is and what its structure looks like, let’s delve deeper into its properties. The next sections explain the main types of claims and properties we can find in JWT payloads and headers and how we use them.

### 11.3.1 Understanding the JWT header

JWTs contain a header that describes the type of token, as well as the algorithm and the key used to sign the token. JWTs are commonly signed using the HS256 and the RS256 algorithms. HS256 uses a secret to encrypt the token, while RS256 uses a private/public key pair to sign the token. We use this information to apply the right algorithm to verify the token’s signature.

Signing algorithms for JWTs

The two most common algorithms used for signing JWTs are HS256 and RS256. HS256 stands for HMAC-SHA256, and it’s a form of encryption that uses a key to produce a hash.

RS256 stands for RSA-SHA256. RSA (Rivest-Shamir-Adleman) is a form of encryption that uses a private key to encrypt the payload. In this case, we can verify that the token’s signature is correct by using a public key.

You can learn more about HMAC and RSA in David Wong’s *Real-World Cryptography* (Manning, 2021).

A typical JWT header is the following:

{

"alg": "RS256",

"typ": "JWT",

"kid": "ZweIFRR4l1dJlVPHOoZqf"

}

Let’s analyze this header:

* alg—Tells us that the token was signed using the RS256 algorithm
* typ—Tells us that this is a JWT token
* kid—Tells us that the key used to sign the token has the ID ZweIFRR4l1dJlVPHOoZqf

A token’s signature can only be verified using the same secret or key that was used to sign it. For security, we often use a collection of secrets or keys to sign the tokens. The kid field tells us which secret or key to use to sign the token so that we can use the right value when verifying the token’s signature.

Some tokens also contain a nonce field in the header. If you see one of those tokens, chances are the token isn’t for your API server unless you’re the creator of the token and you know what the value for nonce is. The nonce field typically contains an encrypted secret that adds an additional layer of security to the JWT. For example, the tokens issued by the Azure Active Directory to access its Graph API contain a nonce token, which means you shouldn’t use those tokens to authorize access to your custom APIs. Now that we understand the properties of a token’s header, the next section explains how to read the token’s claims.

### 11.3.2 Understanding JWT claims

The payload of a JWT contains a set of claims. Since a JWT payload is a JSON document, the claims come in the form of key-value pairs.

There are two types of claims: *reserved claims*, which are part of the JWT specification, and *custom claims*, which are claims we can add to enrich the tokens with additional information.[**7**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/11.htm#pgfId-1111956) The JWT specification defines seven reserved claims:

* iss *(issuer)*—Identifies the issuer of the JWT. If you use an identity-as-a-service provider, the issuer identifies that service. It typically comes in the form of an ID or a URL.
* sub *(subject)*—Identifies the subject of the JWT (i.e., the user sending the request to the server). It typically comes in the form of an opaque ID (i.e., an ID that doesn’t disclose the user’s personal details).
* aud *(audience)*—Indicates the recipient for which the JWT is intended. This is our API server. It typically comes in the form of an ID or a URL. It’s crucial to check this field to validate that the token is intended for our APIs. If we don’t recognize the value in this field, it means the token isn’t for us, and we must disregard the request.
* exp *(expiration time)*—A UTC timestamp that indicates when the JWT expires. Requests with expired tokens must be rejected.
* nbf *(not before time)*—A UTC timestamp that indicates the time before which the JWT must not be accepted.
* iat *(issued at time)*—A UTC timestamp that indicates when the JWT was issued. It can be used to determine the age of the JWT.
* jti *(JWT ID)*—A unique identifier for the JWT.

The reserved claims are not required in the JWT payload, but it’s recommended to include them to ensure interoperability with third-party integrations.

Listing 11.1 Example of JWT payload claims

{

"iss": "https://auth.coffeemesh.io/",

"sub": "ec7bbccf-ca89-4af3-82ac-b41e4831a962",

"aud": "http://127.0.0.1:8000/orders",

"iat": 1667155816,

"exp": 1667238616,

"azp": "7c2773a4-3943-4711-8997-70570d9b099c",

"scope": "openid"

}

Let’s dissect the claims in listing 11.1:

* iss tells us that the token has been issued by the https://auth.coffeemesh.io server identity service.
* sub tell us that the user has the identifier ec7bbccf-ca89-4af3-82ac-b41e4831a962. The value of this identifier is owned by the identity service. Our APIs can use this value to control access to the resources owned by this user in an opaque way. We say this ID is opaque because it doesn’t disclose any personal information about the user.
* aud tells us that this token has been issued to grant access to the orders API. If the value of this field is a different URL, the orders API will reject the request.
* iat tells us that the token was issued on the 30th of October of 2022 at 6:50 p.m. UTC.
* exp tells us that the token expires on the 31st of October of 2022 at 5:50 p.m. UTC.
* azp tells us that the token has been requested by an application with identifier 7c2773a4-3943-4711-8997-70570d9b099c. This is typically a frontend application. This claim is common in tokens that have been issued using the OpenID Connect protocol.
* The scope field tells us that this token was issued using the OpenID Connect protocol.

Now that we know how to work with token claims, let’s see how we produce and validate tokens!

### 11.3.3 Producing JWTs

To form the final JWT, we encode the header, the payload, and the signature using base64url encoding. As documented in RFC 4648 (<http://mng.bz/aPRj>), base64url encoding is similar to Base64, but it uses non-alphanumeric characters and omits padding. The header, payload, and signature are then concatenated using periods as separators. Libraries like PyJWT take care of the heavy lifting of producing a JWT. Let’s say we want to produce a token for the payload we saw in listing 11.1:

payload = {

"iss": "https://auth.coffeemesh.io/",

"sub": "ec7bbccf-ca89-4af3-82ac-b41e4831a962",

"aud": "http://127.0.0.1:8000/orders",

"iat": 1667155816,

"exp": 1667238616,

"azp": "7c2773a4-3943-4711-8997-70570d9b099c",

"scope": "openid"

}

To produce a signed token with this payload, we use PyJWT’s encode() function, passing in the token, the key to sign the token, and the algorithm we want to use to sign the token:

>>> import jwt

>>> jwt.encode(payload=payload, key='secret', algorithm='HS256')

➥ 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29mZ

➥ mVlbWVzaC5pby8iLCJzdWIiOiJlYzdiYmNjZi1jYTg5LTRhZjMtODJhYy1iNDFlNDgzMWE5

➥ NjIiLCJhdWQiOiJodHRwOi8vMTI3LjAuMC4xOjgwMDAvb3JkZXJzIiwiaWF0IjoxNjY3MTU

➥ 1ODE2LCJleHAiOjE2NjcyMzg2MTYsImF6cCI6IjdjMjc3M2E0LTM5NDMtNDcxMS04Otk3Lt

➥ cwNTcwZDliMDk5YyIsInNjb3BlIjoib3BlbmlkIn0.sZEXZVitCv0iVrbxGN54GJr8QecZf

➥ HA\_pdvfEMzT1dI'

In this case, we’re signing the token with a secret keyword using the HS256 algorithm. For a more secure encryption, we use a private/public key pair to sign the token with the RS256 algorithm. To sign JWTs, we typically use certificates that follow the X.509 standard, which allows us to bind an identity to a public key. To generate a private/ public key pair, run the following command from your terminal:

$ openssl req -x509 -nodes -newkey rsa:2048 -keyout private\_key.pem \

-out public\_key.pem -subj "/CN=coffeemesh"

The minimum input for an X.509 certificate is the subject’s common name (CN), which in this case we set to coffeemesh. If you omit the -subj flag, you’ll be prompted with a series of questions about the identity you want to bind the certificate to. This command produces a private key under a file named private\_key.pem, and the corresponding public certificate under a file named public\_key.pem. If you’re unable to run these commands, you can find a sample key pair in the GitHub repository provided with this book, under ch11/private\_key.pem and ch11/public\_key.pem.

Now that we have a private/public key pair, we can use them to sign our tokens and to validate them. Create a file named jwt\_generator.py and paste into it the contents of listing 11.2, which shows how to generate JWT tokens signed with a private key. The listing defines a function, generate\_jwt(), which generates a JWT for the payload defined within the function. In the payload, we set the iat and the exp properties dynamically: iat is set to the current UTC time; exp is set to 24 hours from now. We load the private key using cryptography’s serialization() function, passing in as parameters the content of our private key file encoded in bytes, as well as the passphrase encoded in bytes. Finally, we encode the payload using PyJWT’s encode() function, passing in the payload, the loaded private key, and the algorithm we want to use to sign the token (RS256).

Listing 11.2 Generating JWTs signed with a private key

# file: jwt\_generator.py

from datetime import datetime, timedelta

from pathlib import Path

import jwt

from cryptography.hazmat.primitives import serialization

def generate\_jwt():

now = datetime.utcnow()

payload = {

"iss": "https://auth.coffeemesh.io/",

"sub": "ec7bbccf-ca89-4af3-82ac-b41e4831a962",

"aud": "http://127.0.0.1:8000/orders",

"iat": now.timestamp(),

"exp": (now + timedelta(hours=24)).timestamp(),

"scope": "openid",

}

private\_key\_text = Path("private\_key.pem").read\_text()

private\_key = serialization.load\_pem\_private\_key(

private\_key\_text.encode(),

password=None,

)

return jwt.encode(payload=payload, key=private\_key, algorithm="RS256")

print(generate\_jwt())

To see this code at work, activate your virtual environment by running pipenv shell, and execute the following command:

$ python jwt\_generator.py

➥ eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29mZm

➥ VlbWVzaC5pby8iLCJzdWIiOiJlYzdiYmNjZi1jYTg5LTRhZjMtODJhYy1iNDFlNDgzMWE5N

➥ jIiLCJhdWQiOiJodHRwOi8vMTI3LjAuMC4xOjgwMDAvb3JkZXJzIiwiaWF0IjoxNjM4MDMx

➥ LjgzOTY5ODczOTEsImV4cCI6MTYzODExOC4yMzk2Otg5OTMsInNjb3BlIjoib3BlbmlkIn0

➥ .GipMvEvZG8ErmMA99geYUq5IkeWpRrnHoViLb1CkRufqC5vgM9555re4IsLLa7yVxNAXIp

➥ FVFBqaoWrloJl6dSQ5r00dvUBSM1EM78KMZ7f0gQqUDFWNoKWCeyQu1QCBzuHTouS4l\_mzz

➥ Ii75Sal3DJLTaj4zr6c\_bQdUuDU1GyrIOJiPSCHSlnKPgg9tjrX8eOcB\_ESGSo9ipnCbPAl

➥ uWp0cDjPRPBNRuiU53sbli-

➥ dTy7WoCD1mXAbqhztwO39kG3DZBkysB4vTnKU4Eul2yNNYK2hHVZQEvAqq8TJjETUS7iekf

➥ 0NSt1qQArJ7cxg6Jh5D7y5pbKmYYsBlFohPg

Now you know how to generate JWTs! The JWT generator from listing 11.2 is handy for running tests, and we’ll use it in the upcoming sections to test our code. Now that we understand how JWTs are generated, let’s see how to inspect their payloads and how to validate them.

### 11.3.4 Inspecting JWTs

Often when working with JWTs you’ll run into validation issues. To understand why a token validation is failing, it’s useful to inspect the payload and verify whether its claims are correct. In this section, you’ll learn to inspect JWTs using three different tools: jwt.io ([https://jwt.io](https://jwt.io/)), the terminal’s base64 command, and with Python. To try out these tools, run the jwt\_generator.py script we created in section 11.3.3 to issue a new token.

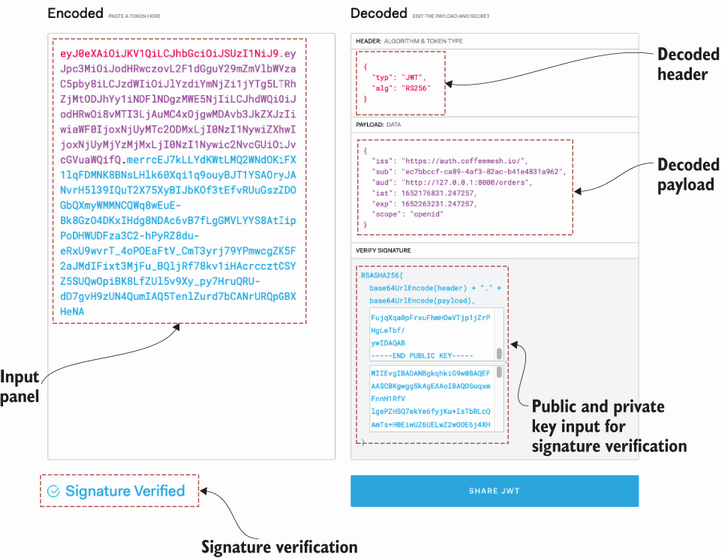


Figure 11.8 jwt.io is a tool that helps you to easily inspect and visualize JWTs. Simply paste the token on the left-side panel. You can also verify the token’s signature by pasting the public key in the VERIFY SIGNATURE box on the right.

jwt.io is an excellent tool that offers an easy way to inspect a JWT. As you can see in figure 11.8, all you need to do is paste the JWT in the input panel on the left. The display panel on the right will show you the contents of the token’s header and payload. You can also verify the token’s signature by providing your public key. To extract the public key from our public certificate, you can use the following command:

$ openssl x509 -pubkey -noout < public\_key.pem > pubkey.pem

This command outputs the public key to a file named pubkey.pem. You need to copy the contents of that file into the public key input panel in jwt.io to verify the token’s signature.

You can also inspect the contents of the JWT by decoding the header and payload in the terminal using the base64 command. For example, to decode the token’s header in the terminal, run the following command:

$ echo eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9 | base64 --decode

{"alg":"RS256","typ":"JWT"}

We can also inspect the contents of a JWT using Python’s base64 library. To decode a JWT header with Python, open a Python shell and run the following code:

>>> import base64

>>> base64.decodebytes('eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9'.encode())

b'{"alg":"RS256","typ":"JWT",}'

Since the JWT payload is also base64url encoded, we use the same methods for decoding it. Now that we know how to inspect JWT payloads, let’s see how we validate them!

### 11.3.5 Validating JWTs

There’re two parts to validating a JWT. On one hand, you must validate its signature, and on the other hand, you must validate that its claims are correct, for example, by ensuring that the token isn’t expired and that the audience is correct. This process must be clear; both steps of the validation process are required. An expired token with a valid signature shouldn’t be accepted by the API server, while an active token with an invalid signature isn’t any good either. Every user request to the server must carry a token, and the token must be validated on each request.

**VALIDATE JWTS ON EACH REQUEST** When a user interacts with our API server, they must send a JWT in each request, and we must validate the token on each request. Some implementations, especially those that use the authorization code flow we discussed in section 11.2.1, store tokens in a session cache and check the request’s token against the cache. That’s not how JWTs are meant to be used. JWTs are designed for stateless communication between the client and the server, and therefore must be validated using the methods we describe in this section.

As we saw in section 11.3.3, tokens can be signed with a secret key or with a private/public key pair. For security, most websites use tokens that are signed with private/public keys, and to validate the signature of such tokens, we use the public key.

Let’s see how we validate a token in code. We’ll use the signing key we created in section 11.3.3 to produce and validate the token. Activate your Pipenv environment by running pipenv shell, and execute the jwt\_generator.py script to issue a new token.

To validate the token, we must first load the public key using the following code:

>>> from cryptography.x509 import load\_pem\_x509\_certificate

>>> from pathlib import Path

>>> public\_key\_text = Path('public\_key.pem').read\_text()

>>> public\_key = load\_pem\_x509\_certificate(public\_key\_text.encode('utf-

➥ 8')).public\_key()

Now that we have the public key available, we can use it to validate a token with the following code:

>>> import jwt

>>> access\_token = "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ..."

>>> jwt.decode(access\_token, key=public\_key, algorithms=['RS256'],

➥ audience=["http://127.0.0.1:8000/orders"])

{'iss': 'https://auth.coffeemesh.io/', 'sub': 'ec7bbccf-ca89-4af3-82ac-

➥ b41e4831a962', 'aud': 'http://127.0.0.1:8000/orders', 'iat':

➥ 1638114196.49375, 'exp': 1638200596.49375, 'scope': 'openid'}

As you can see, if the token is valid, we’ll get back the JWT payload. If the token is invalid, this code will raise an exception. Now that we know how to work with and validate JWTs, let’s see how we authorize requests in an API server.

## 11.4 Adding authorization to the API server

Now that we know how to validate access tokens, let’s put all this code together in our API server. In this section, we add authorization to the orders API. Some endpoints of the orders API are protected, while others must be accessible to everyone. Our goal is to ensure that our server checks for valid access tokens under the protected endpoints.

We’ll allow public access to the /docs/orders and the /openapi/orders.json endpoints since they serve the API documentation that must be available for all consumers. All other endpoints require valid tokens. If the token is invalid or is missing in the request, we must reject the request with a 401 (Unauthorized) status code, which indicates that credentials are missing.

How do we add authorization to our APIs? There’re two major strategies: handling validation in an API gateway or handling validation in each service. An *API gateway* is a network layer that sits in front of our APIs.[**8**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/11.htm#pgfId-1112127) The main role of an API gateway is to facilitate service discovery, but it can also be used to authorize user access, validate access tokens, and enrich the request with custom headers that add information about the user.

The second method is to handle authorization within each API. You’ll handle authorization at the service level when your API gateway can’t handle authorization or when an API gateway doesn’t fit in your architecture. In this section, we’ll learn to handle authorization within the service since we don’t have an API gateway.

A question that often comes up is, where exactly in our code do we handle authorization? Since authorization is needed to validate user access to the service through the API, we implement it in the API middleware. As you can see in figure 11.9, *middleware* is a layer of code that provides common functionality to process all our requests. Most web servers have a concept of middleware or request preprocessors, and that’s where our authorization code goes. Middleware components are usually executed in order, and typically we can choose the order in which they’re executed. Since authorization controls access to our server, the authorization middleware must be executed early.

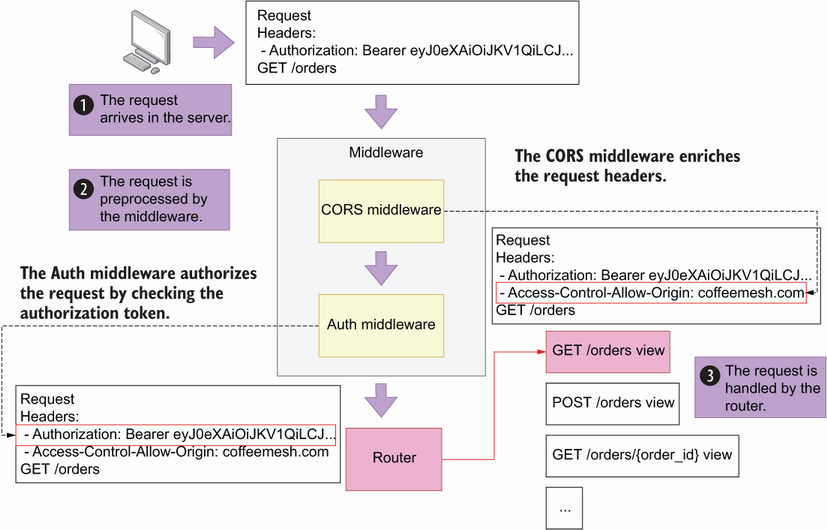


Figure 11.9 A request is first processed by the server middleware, such as the CORS and auth middleware, before making it to the router, which maps the request to the corresponding view function.

### 11.4.1 Creating an authorization module

Let’s first create a module to encapsulate our authorization code. Create a file named orders/web/api/auth.py and copy the code in listing 11.3 into it. We start by loading the public key we created in section 11.3.3. To validate the token, we first retrieve the headers and load the public key. We use PyJWT’s decode() function to validate the token, passing in as parameters the token itself, the public key required to validate the token, the expected list of audiences, and the algorithms used to sign the key.

Listing 11.3 Adding an authorization module to the API

# file: orders/web/api/auth.py

from pathlib import Path

import jwt

from cryptography.x509 import load\_pem\_x509\_certificate

public\_key\_text = (

Path(\_\_file\_\_).parent / "../../../public\_key.pem"

).read\_text()

public\_key = load\_pem\_x509\_certificate(

public\_key\_text.encode()

).public\_key()

def decode\_and\_validate\_token(access\_token):

"""

Validates an access token. If the token is valid, it returns the token payload.

"""

return jwt.decode(

access\_token,

key=public\_key,

algorithms=["RS256"],

audience=["http://127.0.0.1:8000/orders"],

)

Now that we created a module that encapsulates the functionality necessary to validate a JWT, let’s incorporate it into the API by adding a middleware that uses it to validate access to the API.

### 11.4.2 Creating an authorization middleware

To add authorization to our API, we create an authorization middleware. Listing 11.4 shows how to implement the authorization middleware. The code in listing 11.4 goes into the orders/web/app.py file, with the newly added code in bold. We implement the middleware as a simple class called AuthorizeRequestMiddleware, which inherits from Starlette’s BaseHTTPMiddleware class. The entry point for the middleware must be implemented in a function called dispatch().

We use a flag to determine whether we should enable authorization. The flag is an environment variable called AUTH\_ON, and we set it to False by default. Often when working on a new feature or when debugging an issue in our API, it’s convenient to run the server locally without authorization. Using a flag allows us to switch authentication on and off according to our needs. If authorization is off, we add the default ID test for the request user.

Next, we check whether the user is requesting the API documentation. In that case, we don’t block the request since we want to make the API documentation visible to all users; otherwise, they wouldn’t know how to form their requests correctly.

We also check the request’s method. If it’s an OPTIONS request, we won’t attempt to authorize the request. OPTIONS requests are preflight requests, also known as cross-origin resource sharing (CORS) requests. The purpose of a preflight request is to check which origins, methods, and request headers are accepted by the API server, and according to W3’s specification, CORS requests must not require credentials (<https://www.w3.org/TR/2020/SPSD-cors-20200602/>). CORS requests are typically handled by the web server framework.

**DEFINITION** *CORS requests*, also known as preflight requests, are requests sent by the web browser to understand which methods, origins, and headers are accepted by the API server. If we don’t process CORS requests correctly, the web browser will abort communication with the API. Fortunately, most web frameworks contain plug-ins or extensions that handle CORS requests correctly. CORS requests aren’t authenticated, so when we add authorization to our server, we must ensure that preflight requests don’t require credentials.

If it’s not a CORS request, we attempt to capture the token from the request headers. We expect the token under the Authorization header. If the Authorization header isn’t found, we reject the request with a 401 (Unauthorized) status code response. The format of the Authorization header’s value is Bearer <ACCESS\_TOKEN>, so if the Authorization header is found, we capture the token by splitting the header value around the space, and we attempt to validate it. If the token is invalid, PyJWT will raise an exception. In our middleware, we capture PyJWT’s invalidation exceptions to make sure we can return a 401 status code response. If no exception is raised, it means the token is valid, and therefore we can process the request, so we return a call to the next callback. We also store the user ID from the token payload in the request’s state object so that we can access it later in the API views. Finally, to register the middleware, we use FastAPI’s add\_middleware() method.

**WHERE DO JSON WEB TOKENS GO?** JWTs go in the request headers, typically under the Authorization header. An Authorization header with a JWT usually has the following format: Authorization: Bearer <JWT>.

Listing 11.4 Adding an authorization middleware to the orders API

# file: orders/web/app.py

**import os**

from fastapi import FastAPI

**from jwt import (**

**ExpiredSignatureError,**

**ImmatureSignatureError,**

**InvalidAlgorithmError,**

**InvalidAudienceError,**

**InvalidKeyError,**

**InvalidSignatureError,**

**InvalidTokenError,**

**MissingRequiredClaimError,**

**)**

**from starlette import status**

**from starlette.middleware.base import (**

**RequestResponseEndpoint,**

**BaseHTTPMiddleware,**

**)**

**from starlette.requests import Request**

**from starlette.responses import Response, JSONResponse**

**from orders.api.auth import decode\_and\_validate\_token**

app = FastAPI(debug=True)

**class AuthorizeRequestMiddleware(BaseHTTPMiddleware):**  ①

**async def dispatch(**  ②

**self, request: Request, call\_next: RequestResponseEndpoint**

**) -> Response:**

**if os.getenv("AUTH\_ON", "False") != "True":**  ③

**request.state.user\_id = "test"**  ④

**return await call\_next(request)**  ⑤

**if request.url.path in ["/docs/orders", "/openapi/orders.json"]:** ⑥

**return await call\_next(request)**

**if request.method == "OPTIONS":**

**return await call\_next(request)**

**bearer\_token = request.headers.get("Authorization")**  ⑦

**if not bearer\_token:**  ⑧

**return JSONResponse(**

**status\_code=status.HTTP\_401\_UNAUTHORIZED,**

**content={**

**"detail": "Missing access token",**

**"body": "Missing access token",**

**},**

**)**

**try:**

**auth\_token = bearer\_token.split(" ")[1].strip()**  ⑨

**token\_payload = decode\_and\_validate\_token(auth\_token)**  ⑩

**except (**  ⑪

**ExpiredSignatureError,**

**ImmatureSignatureError,**

**InvalidAlgorithmError,**

**InvalidAudienceError,**

**InvalidKeyError,**

**InvalidSignatureError,**

**InvalidTokenError,**

**MissingRequiredClaimError,**

**) as error:**

**return JSONResponse(**

**status\_code=status.HTTP\_401\_UNAUTHORIZED,**

**content={"detail": str(error), "body": str(error)},**

**)**

**else:**

**request.state.user\_id = token\_payload["sub"]**  ⑫

**return await call\_next(request)**

**app.add\_middleware(AuthorizeRequestMiddleware)**  ⑬

from orders.api import api

① We create a middleware class by inheriting from Starlette’s BaseHTTPMiddleware base class.

② We implement the middleware’s entry point.

③ We authorize the request if AUTH\_ON is set to True.

④ If authorization is off, we bind a default user named test to the request.

⑤ We return by calling the next callback.

⑥ The documentation endpoints are publicly available, so we don’t authorize them.

⑦ We attempt to fetch the Authorization header.

⑧ If the Authorization header isn’t set, we return a 401 response.

⑨ We capture the token from the Authorization header.

⑩ We validate and retrieve the token’s payload.

⑪ If the token is invalid, we return a 401 response.

⑫ We capture the user ID from the token’s sub field.

⑬ We register the middleware using FastAPI’s add\_middleware() method.

Our server is ready to start validating requests with JWTs! Let’s run a test to see our authorization code at work. Activate the virtual environment by running pipenv shell, and start the server with the following command:

$ AUTH\_ON=True uvicorn orders.web.app:app --reload

From a different terminal, make an unauthenticated request using cURL (some of the output is truncated) with the -i flag, which displays additional information, such as the response status code:

$ curl -i http://localhost:8000/orders

HTTP/1.1 401 Unauthorized

[...]

{"detail":"Missing access token","body":"Missing access token"}

As you can see, a request with a missing token is rejected with a 401 error and a message telling us that the access token is missing. Now generate a token using the jwt\_ generator.py script we implemented in section 11.3.3, and use the token to make a new request:

curl http://localhost:8000/orders -H 'Authorization: Bearer

➥ eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6ImI3NTQwM2QxLWUzZDktNDgzYy0

➥ 5MjZhLTM4NDRhM2Q4OWY1YyJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29mZmVlbWVzaC5pb

➥ y8iLCJzdWIiOiJlYzdiYmNjZi1jYTg5LTRhZjMtODJhYy1iNDFlNDgzMWE5NjIiLCJhdWQi

➥ OiJodHRwOi8vMTI3LjAuMC4xOjgwMDAvb3JkZXJzIiwiaWF0IjoxNjM4MTE3MjEyLjc5OTE

➥ 3OSwiZXhwIjoxNjM4MjAzNjEyLjc5OTE3OSwic2NvcGUiOiJvcGVuaWQifQ.F1bmgYm1acf

➥ i1NMm5JGkbYQYWFNvG1-7BAXEnIqNdF0th\_DYcnEm\_p3YZ5hQ93v4QWxDx9muit6InKs-

➥ MHqhChP2k6DakpSocaqbgJ\_IHpqNhTaEzByqZjoNfZFyQLZMo3yEaQB8S\_x0LcKOOqeoPYl

➥ GSWM1eAUy7VFBXmvMUZrUj-yoK721U9vevgM-wdVyYFVtpTRuyjCoWMjJEVadNn-

➥ Zrxr0ghlRQnwEx-YdTbbEMkk\_vVLWoWeEgj7mkBE167fr-

➥ fyGUKBqa2F71Zwh8DaDQz79Ph\_STOY6BTlCnAVL8XwnlIOhJWpSHuc90Kynn\_RX49\_yJrQH

➥ KF-xLoflWg'

{"orders":[]}

If the token is valid, this time you’ll get a successful response with a list of orders. Our authorization code is working! The next step is to ensure that users can access only their own resources in the server. Before we do that, though, let’s add one more piece of middleware to handle CORS requests.

### 11.4.3 Adding CORS middleware

Since we’re going to allow interactions with a frontend application, we also need to enable the CORS middleware. As we saw in section 11.4.2, CORS requests are sent by the browser to know which headers, methods, and origins are allowed by the server. FastAPI’s CORS middleware takes care of populating our responses with the right information. Listing 11.5 shows how to modify the orders/web/app.py file to register the CORS middleware, with the newly added code in bold and omitting some of the code in listing 11.5 with an ellipsis.

As we did previously, we use FastAPI’s add\_middleware() method to register the CORS middleware, and we pass along the necessary configuration. For testing purposes, we’re using wildcards to allow all origins, methods, and headers, but in your production environment you must be more specific. In particular, you must restrict the allowed origins to your website’s domain and other trusted origins.

The order in which we register our middleware matters. Middleware is executed in reverse order of registration, so the latest registered middleware is executed first. Since the CORS middleware is required for all interactions between the frontend client and the API server, we register it last, which ensures it’s always executed.

Listing 11.5 Adding CORS middleware

# file: orders/web/app.py

import os

from fastapi import FastAPI

from jwt import (

ExpiredSignatureError,

ImmatureSignatureError,

InvalidAlgorithmError,

InvalidAudienceError,

InvalidKeyError,

InvalidSignatureError,

InvalidTokenError,

MissingRequiredClaimError,

)

from starlette import status

from starlette.middleware.base import RequestResponseEndpoint, BaseHTTPMiddleware

**from starlette.middleware.cors import CORSMiddleware** ①

from starlette.requests import Request

from starlette.responses import Response, JSONResponse

from orders.api.auth import decode\_and\_validate\_token

app = FastAPI(debug=True)

...

app.add\_middleware(AuthorizeRequestMiddleware)

**app.add\_middleware(**

**CORSMiddleware,**  ②

**allow\_origins=["\*"],**  ③

**allow\_credentials=True,**  ④

**allow\_methods=["\*"],**  ⑤

**allow\_headers=["\*"],**  ⑥

**)**

from orders.api import api

① We import Starlette’s CORSMiddleware class.

② We register CORSMiddleware using FastAPI’s add\_middleware() method.

③ We allow all origins.

④ We support cookies for cross-origin requests.

⑤ We allow all HTTP methods.

⑥ We allow all headers.

We’re almost ready! Our server can now authorize users and handle CORS requests. The next step is to ensure each user can only access their data.

## 11.5 Authorizing resource access

We’ve protected our API by making sure only authenticated users can access it. Now we must ensure that the details of each order are only accessible to the user who placed it; we don’t want to allow users to access each other’s data. We call this type of validation *authorization*, and in this section, you’ll learn to add it to your APIs.

### 11.5.1 Updating the database to link users and orders

We’ll start by removing the orders currently present in the database. Those orders are not associated with a user and therefore won’t work once we enforce an association between each order and a user. cd into the ch11 directory, activate the virtual environment by running pipenv shell, and open a Python shell by running the python command. Within the Python shell, run the following code:

>>> from orders.repository.orders\_repository import OrdersRepository

>>> from orders.repository.unit\_of\_work import UnitOfWork

>>> with UnitOfWork() as unit\_of\_work:

... orders\_repository = OrdersRepository(unit\_of\_work.session)

... orders = orders\_repository.list()

... for order in orders: order.delete(order.id)

... unit\_of\_work.commit()

Our database is now clean, so we’re ready to get rolling. How do we associate each order with a user? A typical strategy is to create a user table and link our orders to user records via foreign keys. But does it really make sense to create a user table for the orders service? Do we want to have a user table per service?

No, we don’t want to have a user table per service since it would involve lots of duplication. As you can see in figure 11.10, we want to have just one user table, and that table must be owned by the user service. Our user service is our identity-as-a-service provider, and therefore our user table already exists. Each user has already an ID, and as we saw in section 11.3.1, the ID is present in the JWT payload under the sub field. All we need to do is add a new column to the orders table to store the ID of the user who created the order.

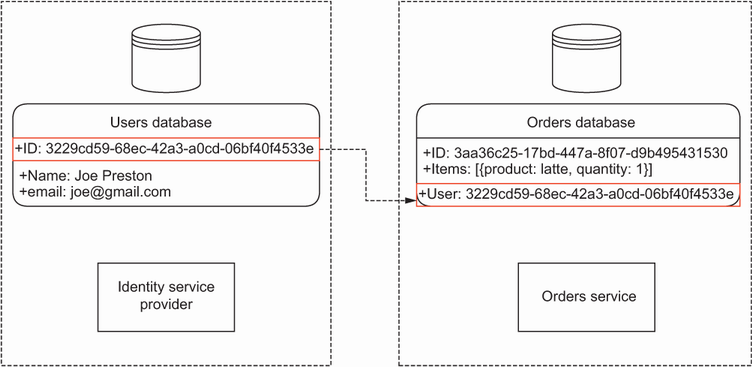


Figure 11.10 To avoid duplication, we keep only one user table under the identity service provider. And to avoid tight coupling between services, we avoid foreign keys between the tables owned by different services.

**LINKING USERS TO THEIR RESOURCES** Two common anti-patterns in microservices architecture is to create one user table per service and to have a shared user table that is directly accessed by multiple services to create foreign keys between users and other resources. Having a user table per service is unnecessary and involves duplicates, while a shared user table across multiple services creates tight coupling between the services and risks breaking them the next time you change the user table’s schema. Since JWTs already contain opaque user IDs under the sub field, it’s good practice to rely on that identifier to link users to their resources.

Listing 11.6 shows how we add a user\_id field to the OrderModel class. The following code goes in the orders/repository/models.py file, and the newly added code is highlighted in bold.

Listing 11.6 Adding a user ID foreign key to the order table

# file: orders/repository/models.py

class OrderModel(Base):

\_\_tablename\_\_ = 'order'

id = Column(String, primary\_key=True, default=generate\_uuid)

**user\_id = Column(String, nullable=False)**  ①

items = relationship('OrderItemModel', backref='order')

status = Column(String, nullable=False, default='created')

created = Column(DateTime, default=datetime.utcnow)

schedule\_id = Column(String)

delivery\_id = Column(String)

① We add a new column called user\_id.

Now that we’ve updated the models, we need to update the database by running a migration. As we saw in chapter 7, running a migration is the process of updating the database schema. As we did in chapter 7, we use Alembic to manage our migrations, which is Python’s best database migration management library. Alembic checks the difference between the OrderModel model and the order table’s current schema, and it performs the necessary updates to add the user\_id column.

**ALTERING TABLES IN SQLITE** SQLite has limited support for ALTER statements. For example, SQLite doesn’t support adding a new column to a table through an ALTER statement. As you can see in figure 11.11, to work around this problem, we need to copy the table’s data to a temporary table and drop the original table. Then we re-create the table with the new fields, copy the data from the temporary table, and drop the temporary table. Alembic handles these operations with its batch operations strategy.

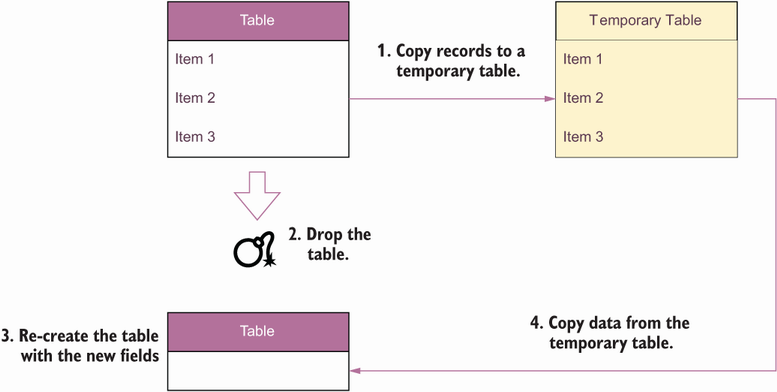


Figure 11.11 When working with SQLite, we use batch operations to make changes to our tables. In a batch operation, we copy data from the original table to a temporary table; then, we drop the original table and re-create it with the new fields; and, finally, we copy back data from the temporary table.

Before we can run the migration, we need to update the Alembic configuration. The change in listing 11.6 adds a new column to the order table, which translates into an ALTER TABLE SQL statement. For local development, we’re working with SQLite, which has limited support for ALTER statements. To ensure that Alembic generates the right migrations for SQLite, we need to update its configuration to run batch operations. *You only need to do this if you work with SQLite*.

To update the Alembic configuration so that we can run the migration, open the migrations/env.py file and search for a function called run\_migrations\_online(). This is the function that runs the migrations against our database. Within that function, search for the following block:

# file: migrations/env.py

with connectable.connect() as connection:

context.configure(

connection=connection,

target\_metadata=target\_metadata

)

And add the following line (highlighted in bold) within the call to the configure() method:

# file: migrations/env.py

with connectable.connect() as connection:

context.configure(

connection=connection,

target\_metadata=target\_metadata,

**render\_as\_batch=True**

)

Now we can generate the Alembic migration and update the database. Run the following command to create the new migration:

$ PYTHONPATH=`pwd` alembic revision --autogenerate -m "Add user id to order table"

Next, we run the migration with the following command:

$ PYTHONPATH=`pwd` alembic upgrade heads

Our database is now ready to start linking orders and users. The next section explains how we fetch the user ID from the request object and feed it to our data repositories.

### 11.5.2 Restricting user access to their own resources

Now that our database is ready, we need to update our API views to capture the user ID when creating or updating an order, or when retrieving the list of orders. Since the changes that we need to make to our view functions are all quite similar, we’ll illustrate how to apply the changes to some of the views. You can refer to the GitHub repository for this book for the full list of changes.

Listing 11.7 shows how to update the create\_order() view function to capture the user ID when placing the order. The newly added code is highlighted in bold. As we saw in section 11.4.2, we store the user ID under the request’s state property, so the first change we make is changing the signature of the create\_order() function to include the request object. The second change is passing the user ID to the OrderService’s place\_order() method.

Listing 11.7 Capturing the user ID when placing an order

# file: orders/web/api/api.py

@app.post(

"/orders", status\_code=status.HTTP\_201\_CREATED, response\_model=GetOrderSchema

)

def create\_order(**request:** **Request,** payload: CreateOrderSchema): ①

with UnitOfWork() as unit\_of\_work:

repo = OrdersRepository(unit\_of\_work.session)

orders\_service = OrdersService(repo)

order = payload.dict()["order"]

for item in order:

item["size"] = item["size"].value

order = orders\_service.place\_order(order**, request.state.user\_id**) ②

unit\_of\_work.commit()

return\_payload = order.dict()

return return\_payload

① We capture the request object in the function signature.

② We capture the user ID from the request’s state object.

We also need to change the OrdersService and the OrdersRepository to ensure they too capture the user ID. The following code shows how to update the OrdersService to capture the user ID:

# file: orders/orders\_service/orders\_service.py

class OrdersService:

def \_\_init\_\_(self, orders\_repository: OrdersRepository):

self.orders\_repository = orders\_repository

def place\_order(self, items, **user\_id**):

return self.orders\_repository.add(items, **user\_id**)

And the following code shows how to update the OrdersRepository to capture the user ID:

# file: orders/repository/orders\_repository.py

class OrdersRepository:

def \_\_init\_\_(self, session):

self.session = session

def add(self, items, **user\_id**):

record = OrderModel(

items=[OrderItemModel(\*\*item) for item in items],

**user\_id=user\_id**

)

self.session.add(record)

return Order(\*\*record.dict(), order\_=record)

Now that we know how to save an order with the user ID, let’s see how we make sure a user gets only a list of their own orders when they call the GET /orders endpoint. Listing 11.8 shows the changes required to the get\_orders() function, which implements the GET /orders endpoint. The newly added code is shown in bold. As you can see, in this case we also need to change the function’s signature to capture the request object. Then we simply pass on the user ID as one of the query filters. No additional changes are required anywhere else in the code since both OrdersService and OrdersRepository are designed to accept arbitrary dictionaries of filters.

Listing 11.8 Ensuring a user only gets a list of their own orders

# file: orders/web/api/api.py

@app.get("/orders", response\_model=GetOrdersSchema)

def get\_orders(

**request: Request**,

cancelled: Optional[bool] = None,

limit: Optional[int] = None

):

with UnitOfWork() as unit\_of\_work:

repo = OrdersRepository(unit\_of\_work.session)

orders\_service = OrdersService(repo)

results = orders\_service.list\_orders(

limit=limit, cancelled=cancelled, **user\_id=request.state.user\_id**

)

return {"orders": [result.dict() for result in results]}

Let’s now turn our attention to the GET /orders/{order\_id} endpoint. What happens if a user tries to retrieve the details of an order that doesn’t belong to them? We can respond with two strategies: return a 404 (Not Found) response indicating that the requested order doesn’t exist, or respond with a 403 (Forbidden) response, indicating that the user doesn’t have access to the requested resource.

Technically, a 403 response is more correct than a 404 when a user is trying to access a resource that doesn’t belong to them. But it also exposes unnecessary information. A malicious user who has valid credentials could leverage our 403 responses to build a map of the existing resources in the server. To avoid that problem, we opt for disclosing less information and return a 404 response. The user ID will become an additional filter when we attempt to retrieve an order from the database.

The following code shows the changes required to the get\_order() function to include the user ID in our queries, with the newly added code in bold. Again, we include the request object in the function signature, and we pass on the user ID to the OrderService’s get\_order() method.

Listing 11.9 Filtering orders with order ID and user ID

# file: orders/web/api/api.py

@app.get("/orders/{order\_id}", response\_model=GetOrderSchema)

def get\_order(**request: Request**, order\_id: UUID):

try:

with UnitOfWork() as unit\_of\_work:

repo = OrdersRepository(unit\_of\_work.session)

orders\_service = OrdersService(repo)

order = orders\_service.get\_order(

order\_id=order\_id, **user\_id=request.state.user\_id**

)

return order.dict()

except OrderNotFoundError:

raise HTTPException(

status\_code=404, detail=f"Order with ID {order\_id} not found"

)

To be able to query orders by user ID as well, we also need to update the OrdersService and the OrdersRepository classes. We’ll change their methods to accept an optional dictionary of arbitrary filters. The OrdersService’s get\_order() method changes like this:

# file: orders/orders\_service/orders\_service.py

def get\_order(self, order\_id, **\*\*filters**):

order = self.orders\_repository.get(order\_id, **\*\*filters**)

if order is not None:

return order

raise OrderNotFoundError(f"Order with id {order\_id} not found")

And the OrdersRepository’s get() and \_get() methods require the following changes:

# file: orders/repository/orders\_repository.py

def \_get(self, id\_, **\*\*filters**):

return (

self.session.query(OrderModel)

.filter(OrderModel.id == str(id\_)).**filter\_by(\*\*filters)**

.first()

)

def get(self, id\_, **\*\*filters**):

order = self.\_get(id\_, **\*\*filters**)

if order is not None:

return Order(\*\*order.dict())

The rest of the view functions in the orders/web/api/api.py file require changes similar to the ones we’ve seen in this section, and the same goes for the remaining methods of the OrdersService and the OrdersRepository classes. As an exercise, I recommend you try to complete the changes necessary to add authorization to the remaining API endpoints. The GitHub repository for this book contains the full list of changes, so feel free to check it out for guidance.

This concludes our journey through API authentication and authorization, and what a journey! You’ve learned what OAuth and OpenID Connect are and how they work. You’ve learned about OAuth flows and when to use each flow. You’ve learned what JWTs are, how to inspect their payloads, and how to produce and validate them. Finally, you’ve learned how to authorize API requests and how to authorize user access to specific resources. You’ve got all you need to start adding robust authentication and authorization to your own APIs!

Appendix C teaches you how to integrate with an identity provider such as Auth0. You’ll also see practical examples of how to use the PKCE and client credentials flows, and you’ll learn to authorize your requests using a Swagger UI.

## Summary

* We authorize access to our APIs using the standard protocols OAuth and OpenID Connect.
* OAuth is an access delegation protocol that allows a user to grant an application access to resources they own in a different website. It distinguishes four authorization flows:
  + *Authorization code*—The API server exchanges a code with the authorization server to request the user’s access token.
  + *PKCE*—The client application, typically an SPA, uses a code verifier and a code challenge to obtain an access token from the authorization server.
  + *Client credentials*—The client, typically another microservice, exchanges a private secret in return for an access token.
  + *Refresh token*—A client obtains a new access token in exchange for a refresh token.
* OpenID Connect is an identity verification protocol that builds on top of OAuth. It helps users easily authenticate to new websites by bringing their identity from other websites, such as Google or Facebook.
* JWTs are JSON documents that contain claims about the user’s access permissions. JWTs are encoded using base64url encoding and are typically signed using a private/public key.
* To authenticate a request, users send their access tokens in the request’s Authorization header. The expected format of this header is Authorization: Bearer <ACCESS\_TOKEN>.
* We use PyJWT to validate access tokens. PyJWT checks that the token isn’t expired, that the audience is correct, and that the signature can be verified with one of the available public keys. If the token is invalid, we reject the request with a 401 (Unauthorized) response.
* To link users to their resources, we use the user ID as represented in the sub claim of the JWT.
* If a user tries to access a resource that doesn’t belong to them, we respond with a 403 (Forbidden) response.
* OPTIONS requests are known as CORS requests or preflight requests. CORS requests must not be protected by credentials.

**1** The issue was reported first by Brian Krebs, “USPS Site Exposed Data on 60 Million Users,” KrebsOnSecurity, November 21, 2018, <https://krebsonsecurity.com/2018/11/usps-site-exposed-data-on-60-million-users/>.

**2** Bill Doerfeld, “API Attack Traffic Grew 300+% In the Last Six Months,” *Security Boulevard*, July 30, 2021, <https://securityboulevard.com/2021/07/api-attack-traffic-grew-300-in-the-last-six-months/>.

**3** Joe Galvin, “60 Percent of Small Businesses Fold Within 6 Months of a Cyber Attack,” *Inc.*, May 7, 2018, <https://www.inc.com/joe-galvin/60-percent-of-small-businesses-fold-within-6-months-of-a-cyber-attack-heres-how-to-protect-yourself.html>.

**4** <https://oauth.net/> is a pretty good website with tons of resources to learn more about the OAuth specification.

**5** N. Sakimura, J. Bradley, and N. Agarwal, “Proof Key for Code Exchange by OAuth Public Clients,” IETF RFC 7636, September 2015, <https://datatracker.ietf.org/doc/html/rfc7636>.

**6** The full specification for how JSON Web Tokens should be produced and validated is available under J. Jones, J. Bradley, and N. Sakimura, “JSON Web Token (JWT),” RFC-7519, May 2015, <https://datatracker.ietf.org/doc/html/rfc7519>.

**7** You can see a full list of the most commonly used JWT claims under <https://www.iana.org/assignments/jwt/jwt.xhtml>.

**8** See Chris Richardson, “Pattern: API Gateway/Backends for Frontends,” <https://microservices.io/patterns/apigateway.html>.