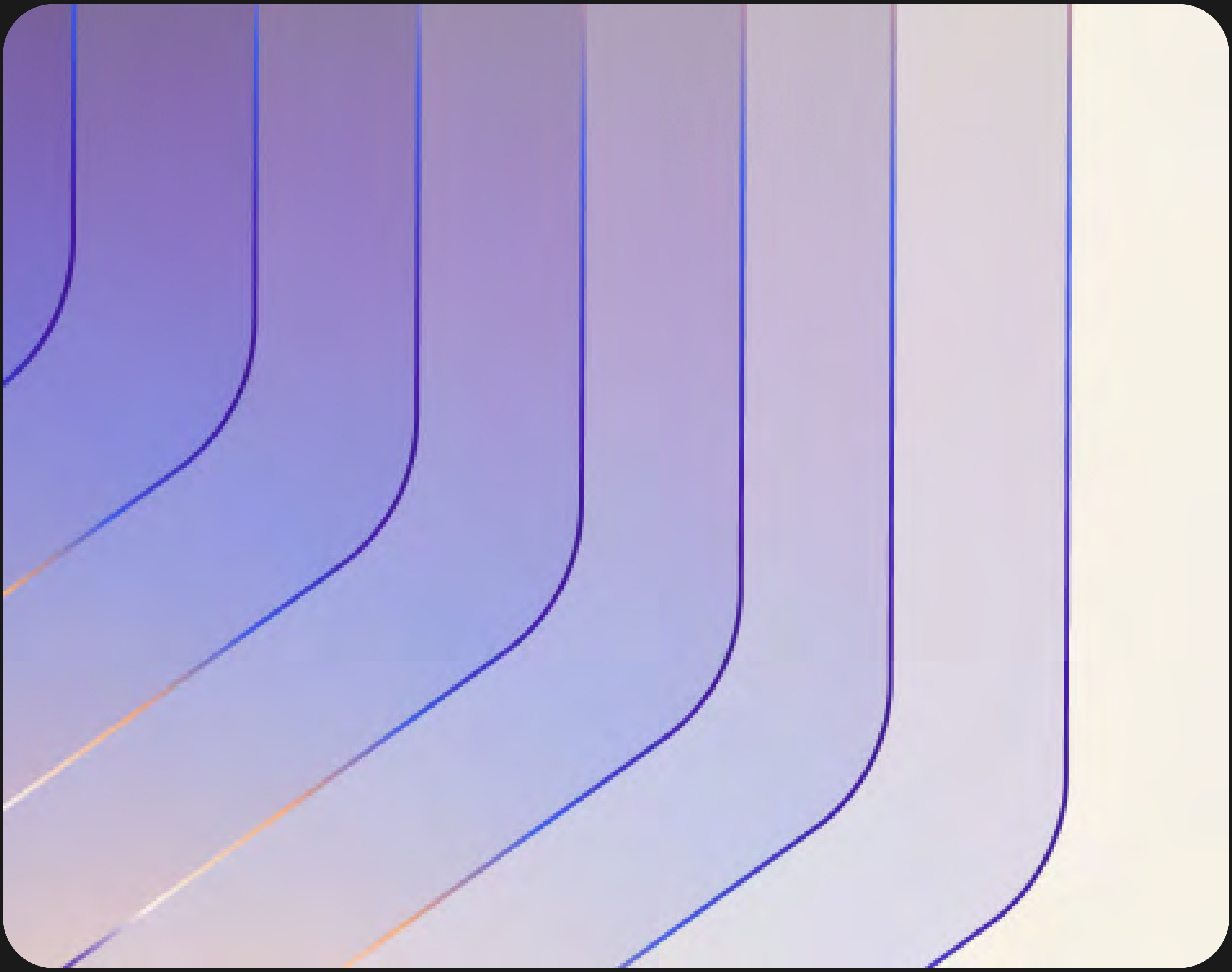
auth0

Whitepaper

OAuth 2.0 and OpenID Connect: The Professional Guide

by Vittorio Bertocci
curated by Andrea Chiarelli



Contents

Preface

I met Vittorio when I joined Auth0 in 2019. I only knew him as an authoritative expert in the field of Identity, and I knew very little about that topic. Usually, he was in charge of onboarding training for the technical staff, but when I went to Bellevue for my onboarding, another trainer was assigned. However, he came to the office to greet me because he had learned an Italian was in the group of new hires. A clear signal that he was more than just an Identity expert: he was a person full of empathy.

I had the honor of working closely with Vittorio at the beginning of my career at Auth0, partly for helping him with this book and partly for reviewing other developer content my team and I created. This gave me the opportunity to learn a lot, not only about the technical aspects of Identity but also about how to explain complex concepts based on the specific audience. His style, rich in metaphors and anecdotes (you will find several in this book), has always fascinated me. His histrionic manner, his marked accent, and his all-Italian gestures establish him in my imagination as a great storyteller.

We started this book project at the end of 2019 and have published it incrementally in two editions. Vittorio had already drawn up the roadmap for the book and shared diagrams and notes with me, but his innumerable commitments led him to postpone its completion.

His untimely death in October 2023 shocked me, as well as the entire community of Identity professionals. Several times, I received requests for information from developers about the book's completion. Therefore, it seemed right to me not to disappoint readers and at the same time honor Vittorio's memory by completing the book according to the original roadmap and based on his notes. I tried as much as possible to maintain his style, even if it was unique and unrepeatable. Since this project started, some things have changed in the meantime, so I had to update some references, even in the already published part. In this work of adaptation, updating, and revision, the help of Aaron Parecki and Filip Skokan was invaluable, and my heartfelt thanks go to them.

One last note before leaving you to read the book. This is not a technical manual about OAuth 2.0 and OpenID Connect. This book will not give you detailed guidelines on how to use OAuth 2.0 and OpenID Connect in your applications. Or rather, you will learn this too, but more than anything else, this guide will explain to you the reason for the small details of these protocols, why we arrived at them, and how we arrived at them.

Understanding the reason for the technical choices and, therefore, the history behind them will help you better understand OAuth 2.0 and OpenID Connect and gain a professional knowledge of them. That said, I hope you will enjoy reading this book!

Andrea Chiarelli

Introduction

This book will help you to make sense of OAuth, OpenID Connect, and the many moving parts that come together to make authentication and delegated authorization happen.

You will discover how authentication and authorization requirements changed in past years and how today's standard protocols evolved and augmented their ancestors to meet those challenges - problems and solutions locked in an ever-escalating arm's race.

You will learn both the whys and the hows of OAuth 2.0 and OpenID Connect. You will learn what parts of the protocol are appropriate to use for each of the classic scenarios and app types (Sign-on for traditional web apps, Single Page Apps, calling API from desktop, mobile, and web apps, and so on). We will examine every exchange and parameter in detail - putting everything in context and always striving to see the reasons behind every implementation choice within the larger picture.

After reading this book, you will have a clear understanding of the classic problems in authentication and delegated authorization, the modern tools that open protocols offer to solve those problems, and a working knowledge of OAuth 2.0 and OpenID Connect. All that will allow you to make informed design decisions - and even to know your way through troubleshooting and network traces.

Chapter 1

Introduction to Digital Identity

In this chapter, you will be able to grasp some of the essentials of identity, both in terms of concepts and the jargon that we like to use in this context. And you'll have a good feeling of the problems, the classic dragons that we want to slay in the identity space, which also happens to be the things that Auth0 by Okta can do for customers.

Without further ado, what is the deal with identity? Why is everyone always saying, "Oh, this is complicated." Why? Just look at the following picture. It is trivially simple: I have just two bodies in here and your basic physics course. It would be one of the easy problems:

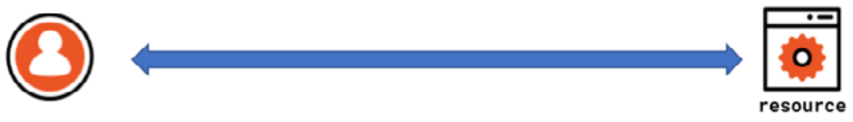


Figure 1.1

I have a resource of some kind, and I have a user — an entity of some kind that wants to access that resource in some capacity. It's just two things doing one action. Why is this so complicated?

Well, for one, there's the fact that this is mission-critical.

When something goes wrong in this scenario, it goes catastrophically wrong. And so, like every mission-critical scenario, of course, it deserves our respect and our attention, and our preparation. There is a lot of energy that goes into preventing this catastrophic scenario from coming true. But in this specific domain of development, the thing that makes this complex is the Cartesian product of all the factors that come into play to determine what you have to do to have a viable solution. Consider the following factors:

- **Resource types**
  Just think of all the types of resources you can have. Just a few years ago, if you'd walk into a bank, you'd have a host, they'd have some central database, and that's it. Today, conversely, pretty much everything is accessible programmatically. So you have the API economy, you have serverless — all those buzzwords actually point to different ways of exposing resources and, of course, websites, apps, and all the things you use in your daily life. Whenever you interact with a computer system, there is a kind of resource that you have to connect to. And, from the point of view of a developer, implementing that connection is actually a lot of work.

- **Development stacks**

  There are minor differences between development stacks that translate into big differences in the code you have to write for implementing access to a resource and the way in which you interact with it. This is one level of complexity.

- **Identities sources**

  The other level of complexity is the sheer magnitude of the sources of identities that you can use today.

  Think of all the ways in which your own identity gets expressed online. You can be a member of a social network, an employee of one company, a citizen of a country. And all of those identities somewhat get expressed in a database somewhere, and that somewhere determines how you pull this information out.

  You connect to Facebook in a certain way. You connect to *Active Directory* in a different way. You get recognized when you're paying your taxes to your country in yet another way. So, again, we encounter another factor of complexity: if you want to extract identity from these repositories, you have to find a way of doing it according to each repository's requirements and characteristics.

- **Client types**

  Finally, there are many more complexity factors, but I just want to mention another one: the incredible richness with which we can consume information today. Think of all the possible clients you can use: from your mobile phone and applications to websites, to your watch. You can literally use anything you want to access the data. And again, these compounds in terms of complexity with the kinds of resources you want to access, the places from where you are extracting information. So, this picture might look simple, but it's all but.

Now, what can Okta do for you to make this a bit more manageable? We offer many different things, but the most salient component of our offering is Auth0. It is a service that you can use for outsourcing most of the authentication functions that you need to have in your solutions - so that you don't have to be exposed to that complexity. In particular, we offer:

- Ways of abstracting away the details of how you connect to multiple sources of identities. Every identity provider will have a different style of doing the identity transactions, and we abstract all of that away from you.

- A way of dealing with the user-management lifecycle. We have user representations and features for dealing with the lifecycle of users and similar.
- A very large number of SDKs and samples, which help you to cross the last mile so that when you're using a particular development stack, you can actually use components to connect to Auth0 in a way that is aligned with the idiom that you're using in that context.
- A degree of customization ability that is absolutely unprecedented in the industry. There is no other service at this point that offers the same freedom you have with Auth0 to customize your experience.

Now, when you need to connect your application to Auth0, you need to do something to tell us, "Auth0, please do authentication". And that something in Auth0 is implemented **using open standards.**

Open standards are agreements, wide consensus agreements that have been crafted by consortiums of different actors in the industry. We identity professionals decided to work on open standards when we came to the realization that everyone - users, customers, and vendors - would have been better off if we had enshrined in common standards, common messages, common protocols, some of the transactions that we know needed to occur when you're doing authentication, and similar. What happened back then is that we went to semi-expensive hotels around the world, met with our peers across the industry, and argued about how applications should present themselves when offering services in the context of an identity transaction. We discussed similar considerations for identity providers. What kind of messages should be exchanged? We literally argued message details down to the semicolon. That's how fun standards authoring is, but it's all worth it: now that we have open standards and all vendors have implemented the open standards, you, as the customer, can choose which vendor you want to use without worries about being locked into a particular technology or vendor. Above all, you can plan to introduce different technologies afterward, without worrying about incompatibilities.

Of course, this is mostly theory: a bit like those simplified school problems disregarding friction or gravity of the moon influencing tides. In reality, there are always little details that you need to iron out. But largely, if you worked in our industry for the last couple of decades, you know that we are so much better off now that we have those open standards we can rely on.

In identity management, you're going to get in touch with many protocols, many of them probably not even invented yet. The ones that are a daily occurrence nowadays are:

- **OpenID Connect**, which is used for signing in.
- **OAuth 2.0**, which is the basis of OpenID Connect and it is a delegation protocol designed to help you access third party APIs.
- **JSON Web Token** or **JWT**, which is a standard token format. Most of the tokens you'll be working with are in this format.
- **SAML**, which is somewhat a legacy (but still very much alive) protocol that is used for doing single sign-on across domains for browsers. SAML also defines a standard token format, which has been very popular in the past and is still very much in use today.

From User Passwords in Every App...

Let’s spend the next few minutes going through a time-lapse-accelerated-whirlwind tour of how authentication technologies evolved. My hope is that by going back to basics and revisiting this somewhat simplified timeline, I'll have the opportunity to show you why things are the way they are today. In doing so, I’ll also have the opportunity to introduce the right terms at the right time. By being exposed to new terminology at the correct time, that is to say, when a given term first arose, you will understand what the corresponding concepts mean in the most general terms. Contrast that with the narrower interpretations of a term’s meaning you’d end up with if you were exposed to it only in the context of solving a specific problem. You might end up thinking that the problem you are solving at the moment is the only thing the concept is good for, missing the big picture and potentially stumbling in all sorts of future misunderstandings. We won’t let that happen!

Let's go back to the absolute basics and think about the scenario I described earlier in Figure 1.1 - the scenario in which I have one resource of some kind, let's say a web application, and a user, and we want to connect the two. Now, what is identity in this context?

We won't get bogged down with philosophy and similar. Identity here can be defined in a very operational, very precise fashion. *We call **digital identity** the set of attributes that define a particular user in the context of a function delivered by a particular application.*

What does it mean? That means that if I am a bookseller, the relevant information I need about a user is largely their credit card number, their shipping address, and the last ten books they bought. That's their digital identity in that context. If I am the tax department, then the digital identity of a user is, again, a physical address, an identifier (here in the USA is the Social Security number), and any other information that is relevant to the motion of extracting money from the citizen. If I am a service that does DNA sequencing, the identity of my user is the username that they use for signing in, their email address for notifications, and potentially their entire genome.

You can see how, for all the various functionalities we want to achieve, we actually have a completely or nearly completely different set of attributes. These might correspond to the same physical person or not. It doesn't matter. From the point of view of designing our systems, that's what the digital identity is. So, you could say that the digital identity of this user is this set of attributes we can place in the application's store. Now, the problem of identity becomes: when do I bring those particular attributes into context? The oldest trick in the world is to have the resource and the user agree on something, such as a shared secret of some sort. So, when the user comes back to the site and presents that secret or demonstrates knowledge of that secret, the website will say, "Okay, I know who you are. You're the same user I saw yesterday". Here is your set of attributes, welcome back. I authenticated the user. In summary, that means grabbing a set of credentials, sending it over, and assuming that those credentials were saved previously in a database. If they match, the user is authenticated. This scenario is summarized in the following picture:

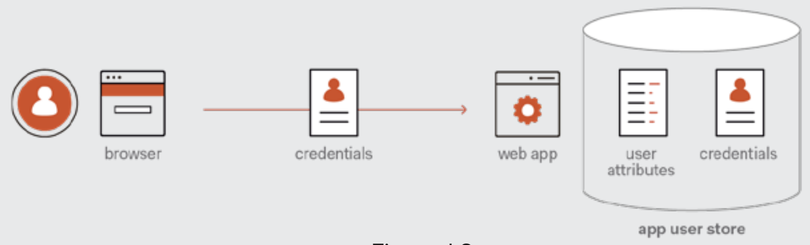


Figure 1.2

Now, you hear a lot of bad things about username and password... and they are all true. That's unfortunate, but it's true. However, it is an extraordinarily simple schema, and as such, it is very, very, very resilient.
Even if we have more advanced technologies, which do more or less the same job, passwords are still very popular. I predict that this year, like every year, someone will say that this is the year in which passwords will die. But I think that passwords will still be around for some time. My favorite metaphor for this is what happens in the natural world. Humans are allegedly the pinnacle of evolution. However, there are still plenty of jellyfish in the sea. They are so simple, and sure, we are more advanced, but I am ready to bet that there are more individual jellyfish than there are humans. The fact that their body plan is simple doesn't mean that it is not successful. You'll see, as we go through this history, that passwords are somewhat building blocks on which more advanced protocols layer on top of. Again, I'm not discounting the efforts of eliminating passwords and using something better, but I'm just trying to set expectations that it's still going to take some time.

...to Directories

Let's make things a bit more interesting. Imagine the scenario in which we have one user and one application. Now, extend this scenario to the situation in which this user is an employee of some company. There is a collection of applications being used by this particular user in the context of the company's business. Most applications are all part of what the user does in the context of their employment. Imagine that one application is for expense notes, the other is for accounting, the other is for warehouse management. Anything you can think of. A few years ago, what happened was that we had a bunch of apps on a computer. Then, we had someone showing up with a coaxial cable, installing token ring networks, and placing all these computers in the network. But that alone didn't make the environment, and in particular the applications, automatically network-ready. What happened is that you'd have exactly the situation - the big thing here - in which you'd have a user accessing different independent apps that knew nothing about each other and that replicated all the functionality that could have been easily centralized. In particular, every user had different usernames and passwords - or I should say different usernames, because, of course, people reuse their passwords. Every time users went to a new app, they had to enter their credentials.

And whenever a user had to leave the company, willingly or not, the administrator had to go on a pilgrimage on all these various apps, run after the user's entries in there, and deprovision them by hand, which, of course, is a tedious and error-prone flow. It's difficult. You often hear horror stories of disgruntled employees using procurement systems to buy large amounts of items just to get back at their former bosses and being able to do so because their credentials in the procurement system weren't timely revoked.

That wasn't a great situation, to say the least.

What happened is that the industry responded by introducing a new entity, which we call the ***directory***. The directory is still extremely popular. It is a software component, a service, which centralizes a lot of the functionalities that you see in Figure 1.3.

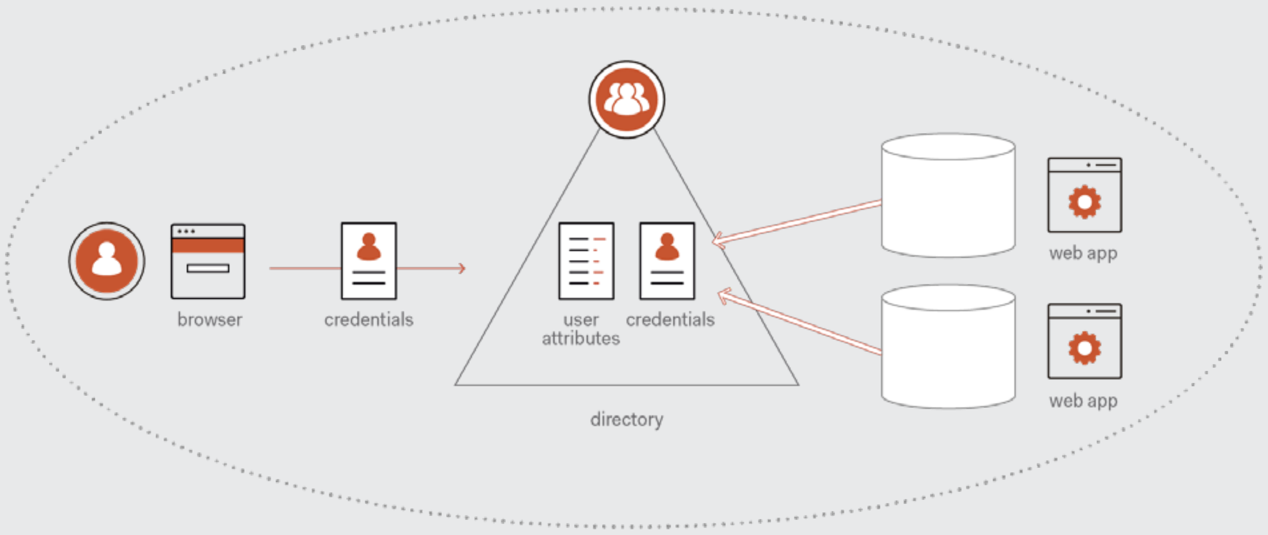


Figure 1.3

Basically, the directory centralized credentials and attributes and made it redundant for applications to implement their own identity management logic. At this point, users would simply sign in with their own central directory, and from that moment onward, they'd have Single Sign-On access to all the other applications. The application developers didn't actually have to code anything for identity to achieve that result. In fact, now that the network infrastructure itself provided the identity information, administrators could now take advantage of this centralized place to deal with the user lifecycle. It can be said that the introduction of the directory is what truly created identity administrators as a category of professionals. The ubiquitous availability of directories created an ecosystem of tooling that helps people run operations, identities, and similar. So, a fantastic improvement - which was predicated on the perimeter. In order for all this to work as intended, you had to have all the actors within that perimeter. The perimeter was often the office building itself, with users actually walking in the building, sitting in front of a particular physical device, and having direct "line of sight" with this cathedral in the center of the enterprise: the directory, a central place knowing everything about everyone.

Cross-Domain SSO

Of course, we know from current business practices that this approach doesn't scale. It works well when you are within one company, but there are so many business processes that require having more than one company.

Think of a classic supplier or reseller. Any of those relationships requires spanning multiple organizations. And so what happens is that when you have a user in one organization that needs to access a different resource in a different organization, you have a problem. In fact, this user does not exist in the resource side directory.

The first way the industry tried to give a solution to this problem was to introduce what we call **shadow accounts**, which means provisioning the user to the resource side directory. This is completely unsustainable, as it presents the same problems we mentioned earlier at a different scale when every application handled identity explicitly. Let's say that we have a user whose lifecycle is managed in one place, their own home directory, but that has been provisioned as an entry in the resource side directory as well. When the user is deprovisioned from their home directory, then there might be a trail of user accounts provisioned in other directories (such as the resource side directory in our scenario) that are still around and that need to be manually deprovisioned. That's, of course, a big problem because the deprovisioning isn't likely to happen timely or, like any changes in general, is harder to reflect in distributed systems that are not centrally managed. Plus, imagine the complexity of having this company, which may be a reseller for many other companies, but needs to somewhat duplicate the work that its customer companies are already doing in their own directories for managing their own users. It's just not sustainable.

So, what happened was that, just like it's classic in computer science, we solved this problem by adding a level of abstraction. We took the capabilities we have seen for the local directory case, and we just abstracted it away. We provided the same transactions, but we described them in a way that is not dependent on network infrastructure. For example, *Active Directory*, and directories in general, rely on an authentication protocol called Kerberos, which is very much integrated with a network layer, hence has specific network hardware requirements. Whereas, of course, in this case of scenarios spanning multiple companies, we have to cross the chasm of the public Internet and cannot afford to impose any requirements as requests will traverse unknown network hardware.

What happened was that the big guys of that time, Sun, IBM, and others, sat at one table and came up with this protocol called **SAML**, which stands for *Security Assertion Markup Language*. In a nutshell, the protocol described a transaction in which a user can sign in in one place and then show proof of signing in in another place and gain access. Here's how it works. We need something that facades my actual resource with some software capable of talking with that protocol, which in this particular case is going to be what we call a *middleware*: a component that stands between your application and the caller, intercepting traffic and executing logic before the requests reach the actual application. Similar protocol capabilities would be exposed on the identity provider side. In the topology shown in Figure 1.3, we have the machine already fulfilling the local directory duties (what we call the domain controller in the directory jargon). We just teach that machine to speak a different language, SAML, which can be considered somewhat of a trading language that we can use for communication outside the company's perimeter.

In order to close this transaction, what happens is that we need to introduce another concept: **trust**. Think of the scenario we were describing earlier, the one within one single directory: in it, every application and every user implicitly believes and trusts the domain controller. The network software itself, whenever you need to authenticate, will send you back to the domain controller, and the domain controller will do its authentication. It is just implicit, it's as natural as the air you're breathing because there is only one place that can perform authentication duties in the entire network.

Now, look at this particular scenario:

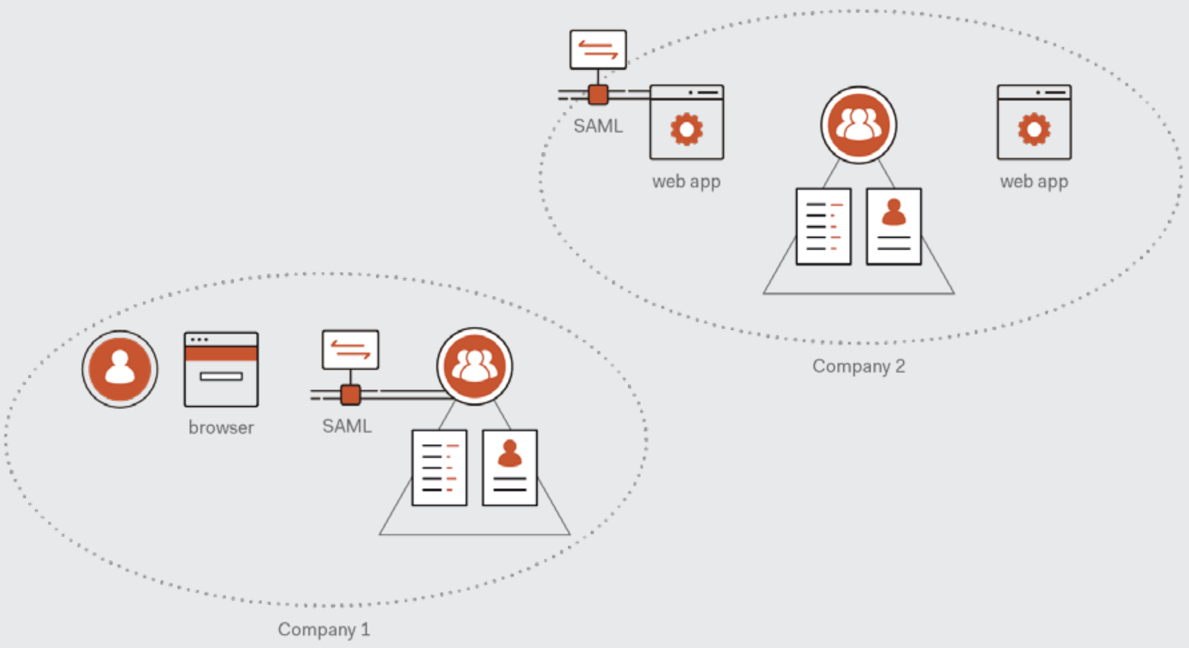


Figure 1.4

The application within the Company 2 perimeter can be accessed by any of its business partners: there is now a choice about from where we want to get users' identities, and there is no longer an obvious default users' source. We say that a *resource trusts an identity provider or an authority when that resource is willing to believe what the authority says about its users*. If the authority says, “This user is one of my users and successfully authenticated five minutes ago”, then the resource will believe it. That's all trust means.

When you set up your middleware in front of your application, you typically configure it with the coordinates of the identity providers that you trust. How does that come into play when you actually make a transaction? Let's see how this works in an actual flow by describing in detail each numbered step shown in the following figure:

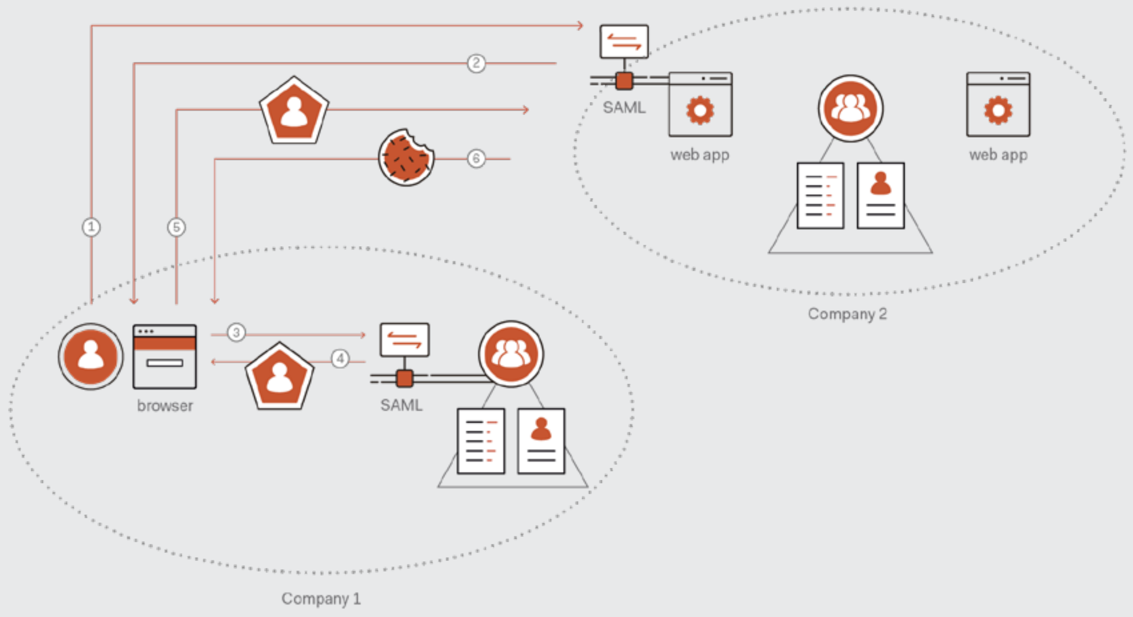


Figure 1.5

In the first leg of the diagram, the user points the browser to the application and attempts to GET a page **(1)**. The middleware in front of the application intercepts the request, sees that the user is not authenticated, and turns the request into an authentication request to the identity provider (*IdP*), as it is configured as one of the trusted IdPs **(2)**.

In concrete terms, the middleware will craft some kind of message, probably a URL with specific query string parameters, and will redirect the browser against one particular endpoint associated with the identity provider **(3)**.

In this particular scenario, the target endpoint belongs to a local identity provider. You can see that the call to the IdP authentication endpoint is occurring within the boundaries of the enterprise. That means that that call will be authenticated using Kerberos, like any other call on the local network. You can already see these layering of protocols, one on top of the other. Thanks to the use of Kerberos and the fact that the user is already authenticated with the local directory, the user will not have to enter any credentials during this call.

Next, the identity provider establishes that the user is already correctly authenticated and determines that the resource is one of the resources that have been recorded and approved. Because of those positive checks, the IdP issues what we call a **security token (4)** to the user. A security token is an artifact, a bunch of bits, used to carry tangible proof that the user has successfully authenticated. Security tokens are digitally signed. What does it mean? A digital signature is something that protects bits from tampering. Let's say that someone modifies any of those bits in transit: when the intended recipient tries to check the signature, it will find that the signature does not compute. The recipient will know for sure that those bits have been modified in transit.

This property is useful for two reasons. One reason is that given that we use public-key cryptography, we expect that the private key used to perform the signature is only accessible by the intended origin of this token. No one else in the universe can perform with that signature, but that particular party. Remember what we just said about trust: that property can be used as proof that a token is coming from a specific entity, and in particular, whether it is a trusted one. The second reason is that given that the token content cannot be modified in transit without breaking the signature, I can use tokens as a mechanism to provide the digital identity of a user on the fly. Instead of having to negotiate in advance the acquisition of the attributes that define the user (the user identity, according to our definition), as an application, I can receive those attributes just in time, together with the token. This might be the first and the last time that this particular user accesses this application, but thanks to the fact that there is trust between the two organizations, I didn't need to do any pre-provisioning steps.

The attributes that travel inside tokens are called claims. A **claim** is simply an attribute packaged in a context that allows the recipient to decide whether to believe that the user indeed possesses that attribute. Think about what happens when boarding a plane. If I present my passport to the gate agents, they will be able to compare my name (as asserted by the passport) with the name printed on my boarding pass and decide to let me go through. The gate agents will reach that conclusion because they trust the government, the entity that issued my passport. If I’d pull out a post-it with my name jolted down with my scrawny chicken legs handwriting and present it to the gate agents in lieu of the passport, I'm probably not going to board the plane - in fact, I'm likely going to be in trouble. The medium truly is the message in this case. The token really does carry the potential to decide whether you trust that particular information or not. Attributes inside tokens become claims. It is an important difference.

Once the identity provider issues a SAML token, it typically returns it to the browser inside an HTML form, together with some JavaScript that triggers as soon as the page is loaded - POSTing the token to the application, where it will be intercepted by the middleware (**5**). The middleware looks at the token, establishes whether it's coming from a trusted source, establishes whether the signature hasn't been broken, etc., etc., and if it's happy with all that, it emits what we call a **session cookie (6)**. The session cookie represents the fact that successful authentication occurred. By setting a cookie to represent the session, the application will be spared from having to do the token dance again for every subsequent request. The session cookie is simply used to enable the application to consider the user authenticated every time the application receives a postback.

This is how SAML solved the particular problem of cross-domain single sign-on. We’ll see that this pattern of exchanging a token for a cookie will also occur with OpenID Connect.

The Password Sharing Anti-Pattern

All this happened in the business world, but the consumer world also didn't stay still from the identity perspective. One thing that happened was that, as we got more and more of our lives online, we found ourselves more and more often with the need to access resources that we handle in a certain application... from a different application.

Let me make a very concrete example. I guess that many of you have LinkedIn, and many of you also have Gmail. Imagine the following scenario. Say that a user is currently already signed in to LinkedIn in whatever way they want. The mechanics of how they got signed in to LinkedIn are not the point in this scenario. Say that LinkedIn wants to suggest you invite all of your Gmail contacts to become part of your LinkedIn network.

We are using LinkedIn and Gmail only because they are familiar names with familiar use cases, but we are in no way implying that they are really implemented in this way nor that they played any direct role in authoring this book.

Now, how was LinkedIn used to do this? I'm using LinkedIn as an example here, but it's basically the behavior of any similar service you can think of before the rise of delegated authorization. Let's take a look at this flow by following the steps in the following figure:

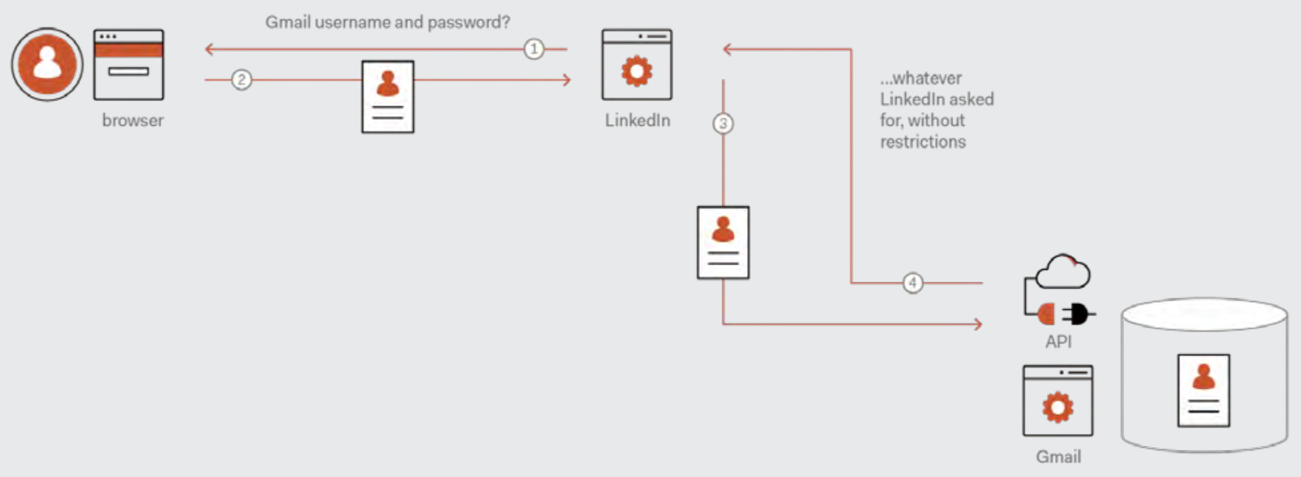

Gmail username and password?
1
2
browser
LinkedIn
3
...whatever LinkedIn asked for, without restrictions
4
API
Gmail

Figure 1.6

LinkedIn would actually ask you for your Gmail username and password, which are normally stored and validated by Gmail **(1)**. You provide LinkedIn with your Gmail credentials **(2)**, and then LinkedIn would use them to actually access the Gmail APIs used by the Gmail app itself for programmatic access to its own service **(3)**. This would achieve what LinkedIn wants, which is to call the APIs in Gmail for listing your contacts **(4)** and sending emails on your behalf.

What is the problem with this scenario? Many problems, but two, in particular, are impossible to ignore.

The first problem is that granting access to your credentials to any entity that is not the custodian of those credentials is always a bad idea. That is mostly because those different entities will not have as much skin in the game as the entity that is actually the original place for those credentials. If LinkedIn does not apply due diligence and save those credentials in an insecure place… sure, they'd get bad PR, but it will not be the catastrophe that it would be for Gmail, for which the user access is now impacted. For example, Gmail users will need to change passwords, creating a situation where they are highly likely to defect or at least to experience lower satisfaction with the service.

Here's the second bad thing. Although LinkedIn's intent with this transaction was good (it is mutually beneficial both for me as a user and for LinkedIn as a service to expand my network), the way in which they have implemented the function gives them way too much power. LinkedIn can actually use this username and password to do whatever they want with my Gmail. They can read my emails, they can delete emails selectively, they can send other emails, they can do everything they want beyond the scenario originally intended - and that's clearly not good.

Delegated Authorization: OAuth 2.0

In response to the challenges outlined at the end of the preceding section, the industry came up with a way of working around the problem of giving too much power to applications.

OAuth 2.0[1] was designed precisely to implement the delegated access scenario described earlier, but without the bad properties that we identified as part of the brute force approach. The defining feature of the OAuth 2.0 approach lies in the introduction of a new entity, the **authorization server**, which explicitly handles operations related to delegated authorization. I won't go too much into the details right now because I'm going to bore you to death about it later in this book.

[1] The first incarnation of OAuth was OAuth 1.0, a protocol that resolved the delegated access scenario but had several limitations and complications. The industry quickly came up with an evolution, named OAuth 2.0, which solved those problems and completely supplanted OAuth1 for all intents and purposes. For that reason, in this text we only discuss OAuth 2.0.

Suffice it to say here that the authorization server has two endpoints:

- The **authorization endpoint**, designed to deal with the interaction with the end-user. It's designed to allow the user to express whether they want a certain service to access their resources in a certain fashion. The authorization endpoint handles the interactive components of the delegated authorization transaction.
- The **token endpoint**, which is designed to deal with software-to-software communication and takes care of actually executing on the intent that the user expressed in terms of permission, consent, delegation, and similar concepts. More details later on.

Please note that in the following discussion, we assume that the user is already signed in to LinkedIn even before the described scenario plays out. We don't care how the sign-in occurred in this context; we just assume it did. OAuth 2.0, as you will hear over and over again, is not a sign-in protocol.

Let's say that, as part of their LinkedIn session, the user gets to a point in which LinkedIn wants to gain access to Gmail API on their behalf, as described in the last section for the analogous scenario.

In the OAuth 2.0 approach, that means that LinkedIn will cause the user to go to Gmail and grant permission to LinkedIn to see their contacts and send mail on their behalf. Let's follow this new flow by taking a look at this figure:

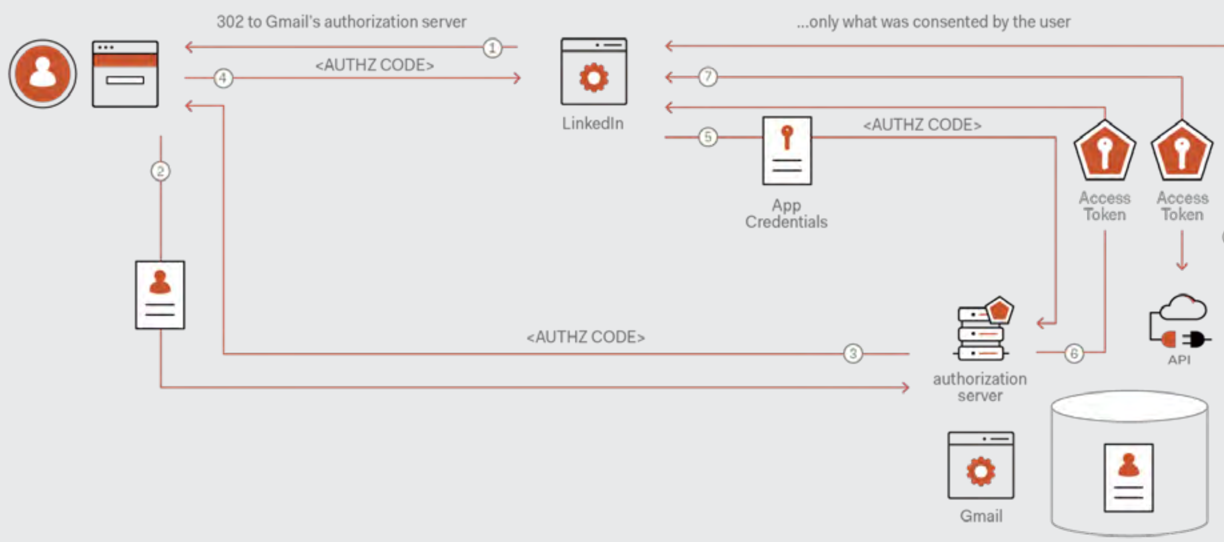


Figure 1.7

LinkedIn follows the OAuth 2.0 specification to craft an *authorization request* and redirect the user's browser to Gmail's authorization server and, in particular, the authorization endpoint **(1)**.
The authorization endpoint is used by Gmail to prompt the user **(2)** for credentials if they are not currently authenticated with the Gmail web application.

This is all within the natural order of things. In fact, it's Gmail asking a Gmail user for Gmail credentials. So, no foul playing here; everything is fine. As soon as the user is authenticated, the Gmail authorization server will prompt the end-user, saying something along the lines of, "Hey, I have this known client, LinkedIn, that needs to access my own APIs using your privileges. In particular, they want to see your contacts, and they want to send emails on your behalf. Are you okay with it?"

Once the user says okay, presumably, the authorization server emits an authorization code **(3)**. An **authorization code** is just an opaque string that constitutes a reminder for the authorization server of the fact that the user granted consent for those permissions for that particular client. The authorization code is returned to LinkedIn via browser **(4)**. From now on, the rest of the transaction occurs on the server side.

Please note: before any of the described transactions could occur, LinkedIn had to go to the authorization server and register itself as a known client. As part of the client registration operation, LinkedIn received an identifier (called **client id**) and, most importantly, a **client secret**. The client id and client secret will be used to prove LinkedIn's identity as an application in requests sent to Gmail's authorization server, in particular to its token endpoint.

The remainder of the diagram explanation will give you an example of how this occurs. Now that it has obtained an authorization code, LinkedIn will reach out to the token endpoint of the authorization server **(5)** and present with its own credentials (client id and client secret) and the authorization code, substantially saying, "Hey, this user consented for this, and I'm LinkedIn. Can I please get access to the resource I want?"

As an outcome of this, the authorization server will emit a new kind of token, which we call an access token **(6)**. The **access token** is an artifact used to grant LinkedIn the ability to access the Gmail APIs **(7)** on the user's behalf, only within the scope of the permissions that the user consented to **(8).**

This solves the excessive permissions problem described in The Password Sharing Anti-Pattern section. In fact, as long as LinkedIn accesses the Gmail APIs only by attempting operations the user consented to, the requests to the API will succeed. As soon as LinkedIn tries to do something different from the consented operations, like, for example, deleting emails, the endpoint will deny LinkedIn access because the access token accompanying the API call is scoped down to the permissions the user consented to (in our example, read contacts and send emails). **Scope** is the keyword that we use here to represent the permissions a client requested on behalf of the user. This mechanism effectively solved the problem of excessive permissions, providing a way to express and enforce delegated authorization.

What we have described so far is the canonical OAuth 2.0 use case, the one for which the protocol was originally designed. In practice, however, OAuth 2.0 is used all over the place, and it incurs all sorts of abuses, that is, in ways in which OAuth 2.0 wasn't designed to be used. Be on the lookout for those problematic scenarios: every time you hear that some solution uses OAuth 2.0, please think of the canonical use case as described here first. OAuth 2.0 supports many other scenarios, and we will discuss most of them in this book. However, the core intent is as we expressed in the use case we described in this section. Thinking about whether a solution is using OAuth 2.0 in line with the intent expressed here or delving from it significantly is a useful mental tool to verify whether you are dealing with a canonical scenario or need to brace for non-standard approaches.

Layering Sign In on Top of OAuth 2.0: OpenID Connect

Let me give you a demonstration of one particularly common type of OAuth 2.0 abuse. As OAuth 2.0 and delegated authorization scenarios started gaining traction, many application developers decided that they wanted to do more than just call APIs. They wanted it to achieve in the consumer space what we achieved with SAML. They wanted to allow users to sign in to their apps, reusing accounts living in a completely different system. Instantiating this new requirement in the scenario we've been discussing, LinkedIn might like users with a Gmail account to be able to use it to sign in to LinkedIn directly, without the need to create a LinkedIn account. In other words, LinkedIn would just want users to be able to sign up on LinkedIn by reusing their Gmail accounts.

This is a sound proposition because, in many cases, people typically aren't crazy about creating new accounts, new passwords, and similar. So, making it possible to reuse accounts is not a bad idea in itself.

However, OAuth 2.0 was not designed to implement sign-in operations. Most providers only exposed OAuth 2.0 as a way of supporting delegated authorization for their API. They did not expose any proper sign-in mechanism as it wasn't the scenario they were after. That didn't deter application developers, who simply piggybacked on OAuth 2.0 flows to achieve some kind of poor man's signing in. Imagine the delegated authorization scenario described for the canonical OAuth 2.0 flow and imagine it taking place with the user not being previously signed in to LinkedIn. The following picture describes this flow:

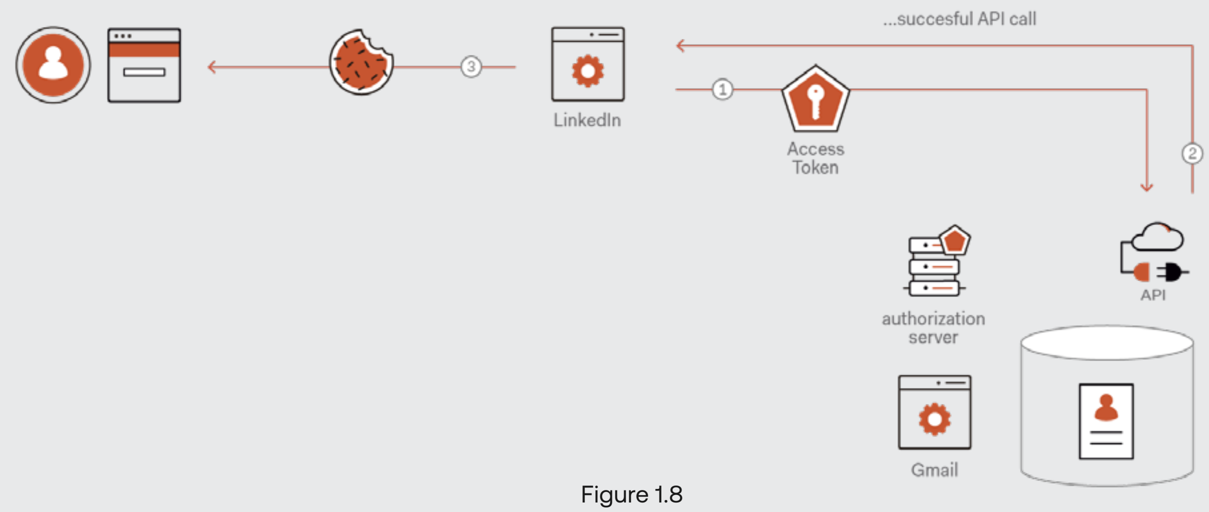


Figure 1.8

LinkedIn can perform the dance to gain access to Gmail APIs without having any authenticated user signed in yet **(1)**. As soon as LinkedIn successfully accesses Gmail APIs **(2)**, it might reason, “Okay, this proves that the person interacting with my app has a legitimate account in Gmail”. So LinkedIn might be satisfied by that and consider this user authenticated, which in practice could be implemented by creating and saving a session cookie **(3)**, as we did during sign-in flows early on when we discussed the SAML approach.

This would be a good time to remind you that we are using LinkedIn and Gmail only because they are familiar names with familiar use cases, but we are in no way implying that they are really implemented in this way.

This pattern for implementing sign-in is still a common practice today. A lot of people do this. It's usually not a good idea, mainly because access tokens are opaque to the clients requesting them, which makes many important details impossible to verify. For example, the fact that an access token can be used to successfully call an API doesn't really say anything about whether that access token was issued for your client or for some other application. Someone could have legitimately obtained that access token via another application (in our scenario, not as LinkedIn, but as some other app) and then somehow managed to inject the token into the request. If LinkedIn just uses that token for calling the API and it reasons, “Okay, as long as I can use this token to call the API without getting an error, I’ll consider the current user authenticated”, then LinkedIn would be fooled in creating an authenticated session.

Another consequence of access tokens being opaque to clients is that an attacker could get a token from a user and somehow inject it into the sign-up operation for a completely different user. Once again, LinkedIn wouldn't know better because unless the API being called returns information that can be used to identify the calling user, the sheer fact that the API call succeeds will not provide any information the client can use to determine that an identity swap occurred.

The attacks I'm describing are called the *Confused Deputy* attack, and they are a classic shortcoming of piggybacking sign-in operations on top of OAuth 2.0.
Even more aggravating: with this approach, there is no way to standardize the OAuth 2.0-based sign-in flow. In our model scenario, the last mile is a successful call to Gmail APIs. If I want to apply the same pattern with Facebook, the last mile would be a successful call to the Facebook Graph APIs, which are dramatically different from the Gmail API. That makes it impossible to enshrine this pattern in a single SDK that can be used to implement sign-in with every provider across the industry, even if they all correctly support OAuth 2.0.

This is where the main players in the industry once again came together and decided to introduce a new specification called OpenID Connect, which formalizes how to layer signing in on top of OAuth 2.0. I'll go into painstakingly fine details about that effort in the rest of the book, but in a nutshell, the central point of the approach is the introduction of a new artifact, which we call the ID token. The ID token can be issued by an authorization server via all the flows OAuth 2.0 defines. OpenID Connect describes how applications can, instead of asking for an access token (or alongside access token requests), ask for an ID token. The following picture summarizes one of such flows:

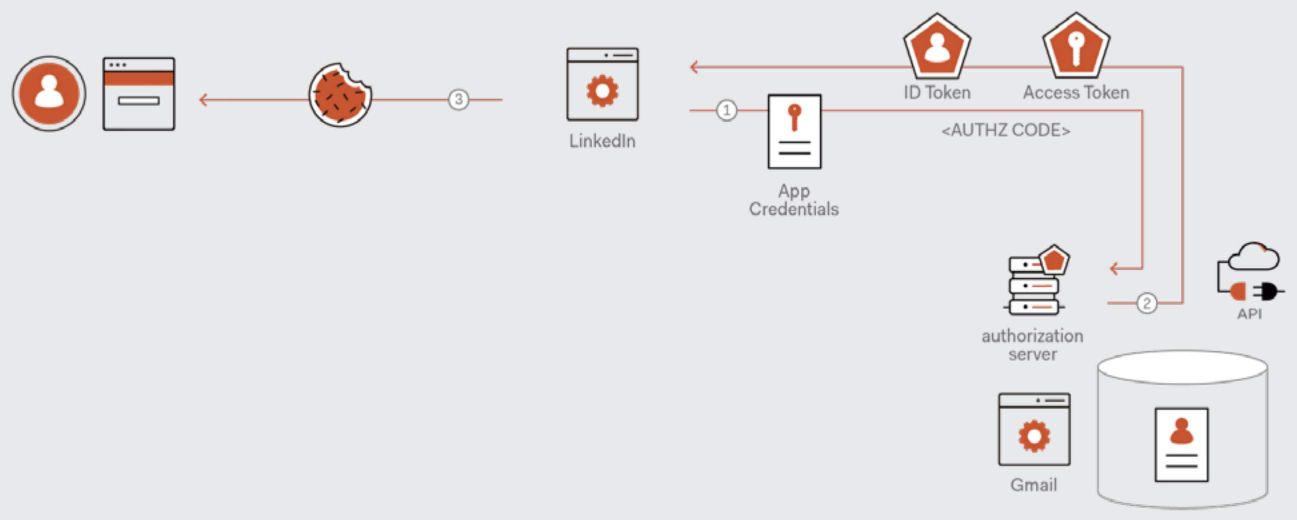


Figure 1.9

An **ID token** is a token meant to be consumed by the client itself, as opposed to being used by the client for accessing a resource. The characteristic of the ID token is that it has a fixed format that clients can parse and validate. Using a known format and the fact that the token is issued for the client itself means that when a client requests and obtains an ID token, the client can inspect and validate it - just like web apps secured via SAML inspect and validate SAML tokens. It also means the ability to extract identity information from it, once again, just like we learned that it's a common practice with SAML. Those properties are what make it possible to achieve proper signing-in using OAuth 2.0. The news introduced by OpenID Connect didn't stop there: the new specification introduced new ways of requesting tokens, including one in which the ID token can be presented to the client directly via the front channel, between the browser and the application. That makes it possible to implement sign-in very easily, just like we have learned in the SAML case, without having to use secrets and a backside integration flow as the canonical OAuth 2.0 API invocation pattern required.

What we have seen in this chapter can be thought of as a rough timeline for the sequence of events that culminated with the creation of OpenID Connect. In the next chapters, we will expand on the high-level flows described here, going deep into the details of the protocol.

Auth0: an Intermediary Keeping Complexity at Bay

What's the role of Auth0 in all this? You can think of Auth0 as an intermediary that has all the capabilities in terms of protocols to talk to pretty much any application that supports the protocols you support, such as OAuth 2.0, OpenID Connect, SAML, WS Federation.

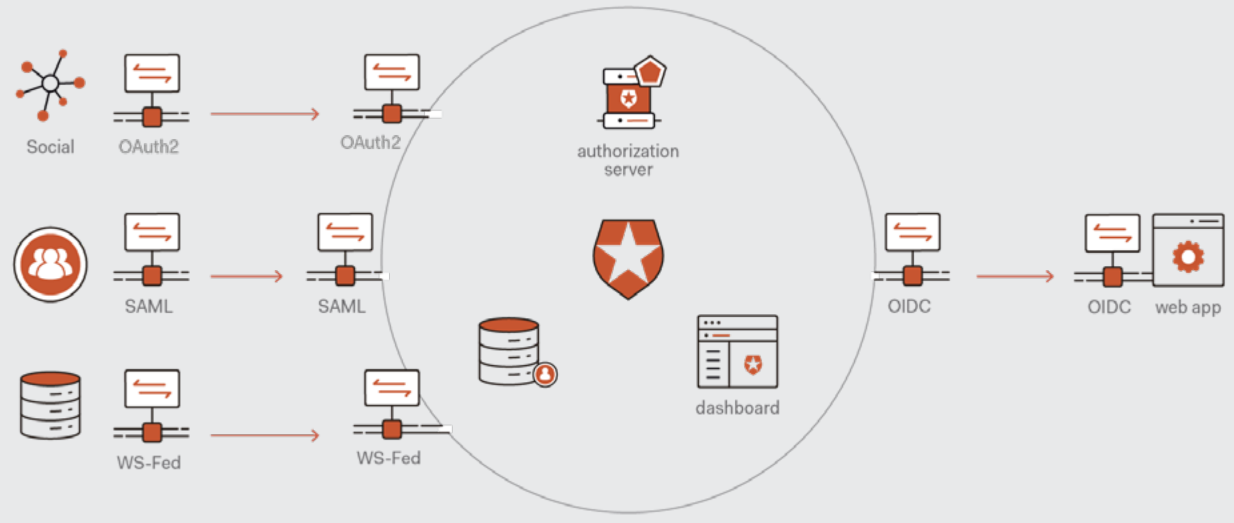


Figure 1.10

You can simply integrate your application with Auth0, which, in a nutshell, is a *super* authorization server, using any of the standard protocol flows we described in this chapter. From that moment on, Auth0 can take over the authentication function. When it’s time to authenticate, your app can redirect users to Auth0 and, in turn, Auth0 will talk to the different identity providers you want to integrate with, in each case using whatever protocol each identity provider requires. If the identity providers of choice use one of the open protocols I mentioned, the integration Auth0 needs to perform is very easy. But if they are using any proprietary approach, for the application developer, it doesn't matter. Once the app redirects to Auth0, Auth0 takes care of the integration details. For you, it's just a matter of flipping a switch and saying, “I want to talk with this particular identity provider” - the result, mediated by Auth0, will always come in the format determined by the open protocol you chose to use for integrating with Auth0. In concrete, that's what we meant earlier when we stated that Auth0 abstracts away the problem from you.

In addition, Auth0 offers a way of managing the lifecycle of a user. Auth0 maintains its own user store; it integrates with external user stores and exposes various operations you can perform to manage users. For example, you can have multiple accounts sourced from various identity providers that accrue to the same account in Auth0 and your app. You can normalize the set of claims you receive from different identity providers so that your application doesn't have to contain any identity provider-specific logic.

We also provide ways of injecting your own code at authentication time so you can easily execute custom logic, such as subscription, billing, or any functionality that just makes sense in your scenario to occur at the same time as authentication.

You have full control over the experience your users will go through, as Auth0 allows you to customize every aspect of the authentication UX. Auth0 makes it very easy for you to use the set of features, mostly by providing a dashboard with a very simple point-and-click interface. You can also use Auth0’s management APIs to achieve programmatic access to everything the dashboard does and more.

That's it for Identity 101. It was a pretty quick whirlwind tour of the last 15 to 20 years of evolution in the world of digital identity. In the next chapters, we'll spend a bit more time sweating the details.

Chapter 2

OAuth 2.0 and OpenID Connect

Let's dig a bit deeper and specifically turn our attention to OAuth and OpenID Connect (OIDC) as protocols.

Have you ever read any of the specifications of those protocols? I am an old hand at this: I was working in this space when there were still CORBA, WS-Trust, and various other old man's protocols. In the past, identity protocols tended to be extraordinarily complicated: they were XML-based and exhibited high-assurance features that made them hard to understand and implement. For example, the cryptography they used supported what was called *message-based security* - granting the ability to achieve secure communications even on plain HTTP. It was an interesting property, but it came at the cost of really intricate message formatting rules that made implementation costs prohibitive for everyone but the biggest industry players.

Now, the new crop of protocols - OAuth, OpenID Connect, and similar - are based on simple HTTP and JSON - a reasonably simple format - and they heavily rely on the fact that everything occurs on secure channels. This simple assumption enormously simplifies things: together with other simplifications and cuts, this makes the new protocols more approachable and at least readable.

However, we are not exactly talking about Harry Potter. Plowing through eighty-six pages of intensely technical language, such as the ones constituting the OpenID Connect Core specification, is a pretty big endeavor, even for committed professionals. If you work in the identity space, you'll find yourself referring to the specifications in detail, over and over again, with a lawyer-like focus, on each and every single word - those documents are dense with meaning. You can also see that the specifications have a pretty high cyclomatic complexity. That's to say, there are multiple links that provide context, and usually, there is not a lot of redundancy. If there is a link pointing to another specification defining a concept used in the current document, you've got to follow the link and actually learn about that concept before you can make any further progress. There's really a very large number of such specifications, even if you limit the scope to just one or two hops from the OpenID Connect and OAuth 2.0 core specs. All the specifications you see in the constellation of OAuth, OpenID, JWT, JWS, and similar are the core, describing the most fundamental aspects that come into play when handling the main scenarios those specifications are meant to address. An entire ring of best practices or new capabilities is not shown here. The complete picture is, in fact, much larger.

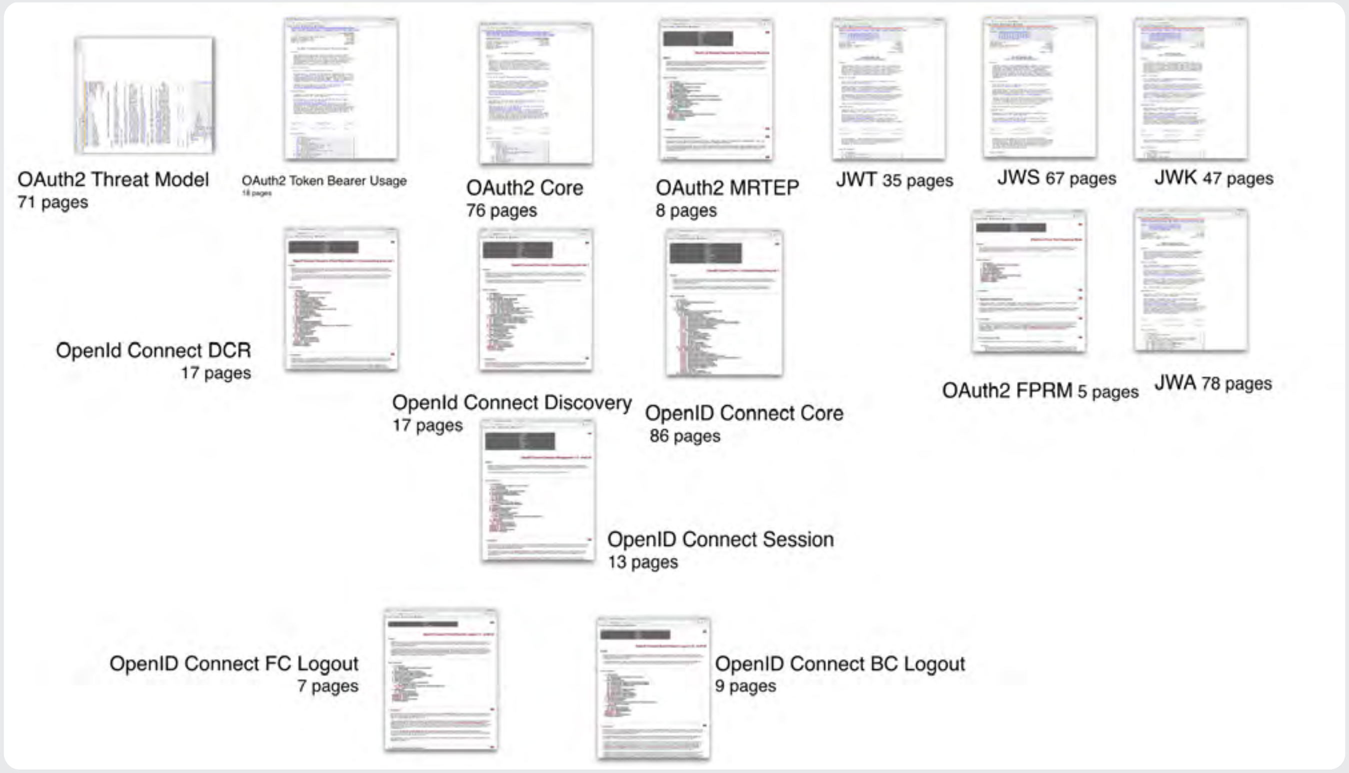

OAuth2 Threat Model 71 pages
OAuth2 Token Bearer Usage 18 pages
OAuth2 Core 76 pages
OAuth2 MRTEP 8 pages
JWT 35 pages
JWS 67 pages
JWK 47 pages
OpenId Connect DCR 17 pages
OpenId Connect Discovery 17 pages
OpenID Connect Core 86 pages
OAuth2 FPRM 5 pages
JWA 78 pages
OpenID Connect Session 13 pages
OpenID Connect FC Logout 7 pages
OpenID Connect BC Logout 9 pages

Figure 2.1

The main reason I am showing you this is to dispel the notion, which a lot of people really like to believe, that adding identity capabilities to one application is just a matter of reading the spec. If you want to do modern identity, just read the OAuth 2.0 and OpenID Connect specifications, and you'll be fine. Of course, the reality is quite different. If that were true, then not many people would be doing modern authentication nowadays.

In fact, reading all these things is **our** job as identity professionals - as the ones who build identity services, SDKs, quick starts, samples, and guides that developers can use to get their job done without necessarily having to be bogged down in the fine-grained details of the underlying protocols. That said, given that the book you are reading is meant to be read by aspiring identity professionals, the fine-grained details of the protocol are among the things we want to learn about - and what you'll find in abundance in the rest of the text.

However, I dislike the classic academic approach, which is so common in other learning material about identity. There, you just get the lecture and a laundry list of the concepts listed in these various specifications -

college style - and are expected to figure out on your own how they apply to your scenarios. The messages, artifacts, and practices defined in those specifications are all there for specific reasons. Typically, it is for addressing use cases and scenarios. Their language is such that it's not usually presented in a scenario-based approach, as it would not be economical in a specification to do so. That's a great approach for formal descriptions and keeping ambiguity to a minimum, but not great for actually understanding how to apply things in concrete.

I'm going to turn things around, and actually, apart from giving you some basic definitions, I want to operate at the scenario level. I want you to understand *why* things are the way they are and how they are applied in particular solutions rather than just asking you to study for a test. In the process, we will eventually end up covering all the main actors and all the main elements in the specifications. Simply, we will not be following the traditional order in which those artifacts are listed in the specs themselves. We'll just follow the order dictated by the jobs to be done
that we want to tackle.

OAuth 2.0 Roles

Let's start with the few definitions I mentioned we need before starting our scenario-based journey through the specifications. OAuth 2.0 and OpenID Connect define a number of primitives required for describing what's going on during identity transactions.

In particular, OAuth 2.0 introduces several canonical **roles** that different actors can play in the context of an identity transaction. As OpenID Connect is built on OAuth 2.0, it inherits those roles as well:

- The first one is the **resource owner**. The resource owner is, quite simply, the user. Think of the LinkedIn and Gmail scenario in the preceding chapter: the resource LinkedIn wants to access is the user's Gmail inbox; hence the user in the scenario is the resource owner.
- Then we have the **resource server**, which is the guardian of the resource, the gatekeeper that you need to clear in order to obtain access. It typically is an API. In our model scenario, the resource server is whatever protects the API that LinkedIn calls for enumerating contacts and sending emails with Gmail on behalf of the resource owner.

- Then, there is the **client**, probably the most salient entity for developers. From the OAuth 2.0 perspective, the client is the application that needs to obtain access to the resource. In our example, that would be the LinkedIn web application.

For OAuth 2.0, which is a delegated authorization protocol and a resource access protocol, every application is modeled as a client. However, we'll see that when we start layering things on top of OAuth 2.0, and for example, we'll use OpenID Connect for signing in, very often what, according to the spec jargon, is called the client will, in fact, be the resource that we want to access. In that sentence, I use “resource” not in the OAuth sense but in the general English language sense of the word. You can see how naming “client” the resource you want to get access to might be confusing!

Now that you have seen in Chapter 1 how OpenID Connect was built on top of OAuth 2.0 scenarios, you know why. That's because in OpenID Connect, signing in means requesting an ID token, which is a special semantic access token meant to be consumed by the requestor itself, rather than for accessing an external resource. Your application is both the client (because it requests the IDtoken) and the resource itself (because it consumes it instead of using it for calling an API), but the term we end up using for describing the app in protocol terms is just client. That can be confusing for the non-initiated, but that's the way it is. I will often highlight this discrepancy throughout the book.

- Finally, we have the **authorization server,** which, as defined in Chapter 1, Introduction to Digital Identity, is the collection of endpoints used for driving the delegated authentication scenarios described there (and many more).

The authorization server exposes the **authorization endpoint**, which is the place where users go for anything that entails interactivity. Practically speaking, the authorization endpoint serves back web pages. It's not always literally the case, as we'll see in the chapter about SPA, but the cases in which we don't show a UI on the authorization point are an exception. The authorization server also features a **token endpoint**, which apps typically speak to programmatically, performing the operation that actually retrieves tokens.

Authorization and token endpoints are defined in OAuth 2.0 Core. OpenID Connect augments those with the **discovery endpoint**. This is a standard endpoint that advertises, in a machine-consumable format, the capabilities of the authorization server. For example, it will list information like the addresses of the two endpoints I just described. Another essential information the discovery endpoint provides is the key that OIDC clients should use for validating tokens issued by this particular authorization server, and so on, and so forth.

OAuth 2.0 Grants and OIDC Flows

The most complicated things in the context of OAuth 2.0 and OpenID Connect are usually what we call **grants**. In a nutshell, grants are just the set of steps a client uses to obtain some kind of credential from the authorization server, for the purpose of accessing a resource. As simple as that. OAuth 2.0 defines a large number of grants because each of them makes the best of the ability of a different client type to connect to the authorization server in their own ways, according to its peculiar security guarantees. Grants also serve the purpose of addressing different scenarios, such as scenarios where access is performed on behalf of the user vs. via privileges assigned to the client itself and many more.

I won't go into details of the various grants here because we are going to pretty much look at all of them inside out through this book. Suffice it to say at this point that there is a core set of grants originally defined by OAuth 2.0: *Authorization Code, Implicit, Resource Owner Credentials, Client Credentials,* and *Refresh Token.* OpenID Connect introduces a new one, the *Hybrid,* which combines two particular OAuth 2.0 grants into one single flow.

In addition to the grants defined by the core OAuth 2.0 and OpenID Connect specifications, the OAuth 2.0 working group at IETF and the OpenID Foundation continuously produce independent extensions devised to address scenarios not originally contemplated by the core specs, or deemed too specific for inclusion. The ability to add new specifications to extend and specialize the core spec is a powerful mechanism that helps the community receive the guidance it needs to address new scenarios as they arise.

The book will examine every essential grant in detail, with a particular emphasis on the scenarios for which a specific grant is most appropriate, the reasons behind the main features characterizing every grant, and the most important factors that need to be taken into account when choosing to solve a scenario with a specific grant.

Chapter 3

Web Sign-In

Starting with this chapter, we are going to dive deeper into concrete scenarios. Let's begin with the most common one: Web Sign-In.

Confidential Clients

Before I actually get into its mechanics, I have to introduce at high level a couple of artifacts and terminology that we use in the context of OAuth 2.0 in OpenID Connect. In particular, I want to talk to you about client types.

A **confidential client** in OAuth 2.0 is a client that has the ability to prove its own application programmatic identity. It's any application to which the authorization server can assign a credential of some type that allows the app to prove its identity as a registered client during any request to the authorization server.
This typically happens with any singleton app. Think of a website that is running on a certain set of machines. Even if executing on a cluster, it's one logical entity running there. When I provision my client by registering it at the authorization server, I have a clear identity for it. I have URLs that determine where this client lives, and I have a flow for getting whatever secret we want to agree upon, which I can save and protect locally.

Allegedly, if the application runs on a server, the server administrator is the only person who can access that secret. Contrast all of this with applications that, for example, run on your device: those apps are all but a singleton. Every phone will have a different instance of Slack, for example. When you download the application from the application store, there is no easy way to get a unique key representing that particular instance of a client.

You certainly cannot embed such a key in the code because it would be de-compiled in a second, and you'd be in trouble. Also, the device is always available in the pockets of the people using it. It is outside of your control, so there is no way for you to protect the key for an extended period of time. A motivated hacker has an infinite time to actually dig into the device, as opposed to a server that must first be breached before it can reveal its secrets.

In summary, confidential clients are clients for which it's appropriate to assign a secret. The classic scenario is websites that run with a server But you can also think of an IoT scenario in which you want to identify the device itself rather than its user.

Another scenario involves long-running processes.
For example, consider a continuous integration system that uses your Jenkins, compiles your product overnight, runs tests, and similar long-running tasks. You'll likely want that daemon to run with its own identity, as opposed to the identity of a user. In fact, if you use the identity of a user and then the user leaves the company, everything may grind to a halt, and no one knows why. This happens because people very often forget that a particular user identity was used for running these scripts. So, assigning its own identity to the daemon is a better option.

One subtlety here is that even if an application is a confidential client, not every single grant that the application does will require using a client credential. It is a capability that the application has, but it doesn't have to exercise it every time. There will be, in fact, scenarios like the one that we are about to explore, in which there is no need to use keys. Typically, the key is used to prove your identity as a client when you're asking for a token to access a different resource. Instead, we'll see that in the case of Web Sign-In, you are the resource.

The *Implicit* Grant with Form Post

The grant we're going to use here is the **Implicit grant with Form Post**. It is kind of a mouthful, but, unfortunately, that's the way the protocol defines it. This is something that wasn't possible before OpenID Connect. It is the easiest way to achieve Web Sign-In using OpenID Connect, and it is really similar to SAML. In fact, it basically follows the same steps that I've described when I demonstrated the first SAML flow in the first chapter, Introduction to Digital Identity.

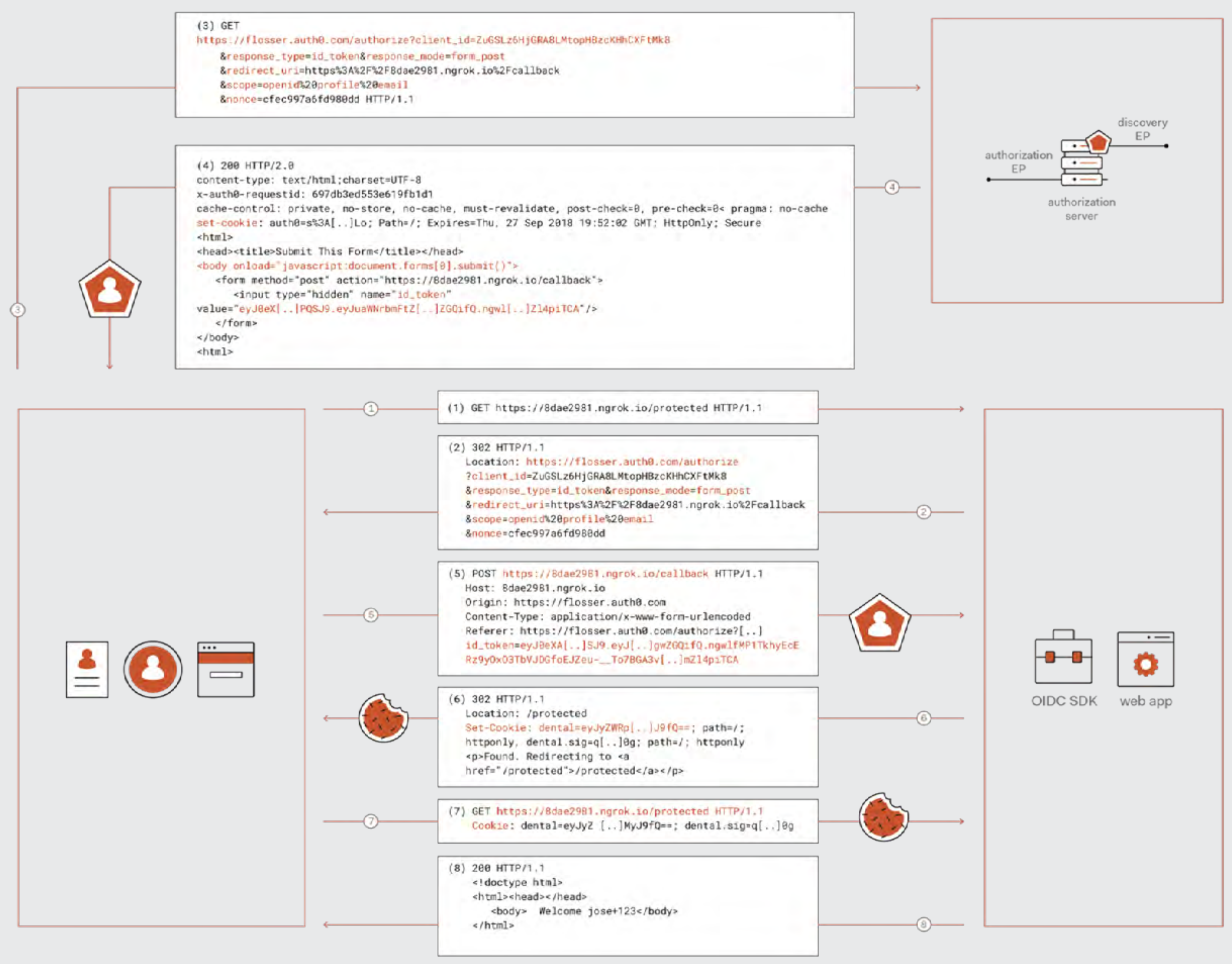
This grant constitutes the basis of something that only OpenID Connect can do, that is, combining signing in to a website with granting that website delegated permission to access an API. What we are going to do now is to study half of that transaction. We'll only look at the sign-in part. When we will talk about APIs, we'll look at the other half. Those two halves can be combined so that the user experience is truly streamlined. Also, in terms of design, combining sign-in and API invocation capabilities makes it possible for an application to play multiple roles. This is a really powerful scenario that wasn't possible before OpenID Connect.

Given that we're using the front channel, we don't need to use the application credentials. There are security implications here and there, but, as just said, it is just like SAML.

Setting this thing up from a developer’s perspective is a thing of beauty. You just install your middleware in front of your application. Then, you use your configuration to point it to the discovery endpoint, as we mentioned in Chapter 2, OpenID Connect and OAuth, and just specify the identifier assigned as a client when you registered your application. In the authorization server, you need to specify the address where you want to get tokens back to the app, and you’re done.

A detailed walkthrough

Let's see in detail how the *Implicit* grant with *form_post* works. Take a look at the scenario shown in Figure 3.1:



```
(3) GET
https://flosser.auth0.com/authorize?client_id=ZuGSLz6HjGRA8LMtopHBzcKHhCXFtMk8
    &response_type=id_token&response_mode=form_post
    &redirect_uri=https%3A%2F%2F8dae2981.ngrok.io%2Fcallback
    &scope=openid%20profile%20email
    &nonce=cfec997a6fd980dd HTTP/1.1
```

```
(4) 200 HTTP/2.0
content-type: text/html;charset=UTF-8
x-auth0-requestid: 697db3ed553e619fb1d1
cache-control: private, no-store, no-cache, must-revalidate, post-check=0, pre-check=0< pragma: no-cache
set-cookie: auth0=s%3A[..]Lo; Path=/; Expires=Thu, 27 Sep 2018 19:52:02 GMT; HttpOnly; Secure
<html>
<head><title>Submit This Form</title></head>
<body onload="javascript:document.forms[0].submit()">
   <form method="post" action="https://8dae2981.ngrok.io/callback">
      <input type="hidden" name="id_token"
value="eyJ0eX[..|PQSJ9.eyJuaWNrbmFtZ[..]ZGQifQ.ngwl[..]Zl4piTCA"/>
   </form>
</body>
<html>
```

```
(1) GET https://8dae2981.ngrok.io/protected HTTP/1.1
```

```
(2) 302 HTTP/1.1
   Location: https://flosser.auth0.com/authorize
   ?client_id=ZuGSLz6HjGRA8LMtopHBzcKHhCXFtMk8
   &response_type=id_token&response_mode=form_post
   &redirect_uri=https%3A%2F%2F8dae2981.ngrok.io%2Fcallback
   &scope=openid%20profile%20email
   &nonce=cfec997a6fd980dd
```

```
(5) POST https://8dae2981.ngrok.io/callback HTTP/1.1
   Host: 8dae2981.ngrok.io
   Origin: https://flosser.auth0.com
   Content-Type: application/x-www-form-urlencoded
   Referer: https://flosser.auth0.com/authorize?[..]
   id_token=eyJ0eXA[..]SJ9.eyJ[..]gwZGQifQ.ngwlfMP1TkhyEcE
   Rz9yOx03TbVJDGfoEJZeu-__To7BGA3v[..]mZl4piTCA
```

```
(6) 302 HTTP/1.1
   Location: /protected
   Set-Cookie: dental=eyJyZWRp[..]J9fQ==; path=/;
   httponly, dental.sig=q[..]0g; path=/; httponly
   <p>Found. Redirecting to <a
   href="/protected">/protected</a></p>
```

```
(7) GET https://8dae2981.ngrok.io/protected HTTP/1.1
    Cookie: dental=eyJyZ [..]MyJ9fQ==; dental.sig=q[..]0g
```

```
(8) 200 HTTP/1.1
    <!doctype html>
    <html><head></head>
       <body>  Welcome jose+123</body>
    </html>
```

Figure 3.1

We have a user with a browser, a web application protected by a middleware implementing OpenID Connect, and an authorization server.

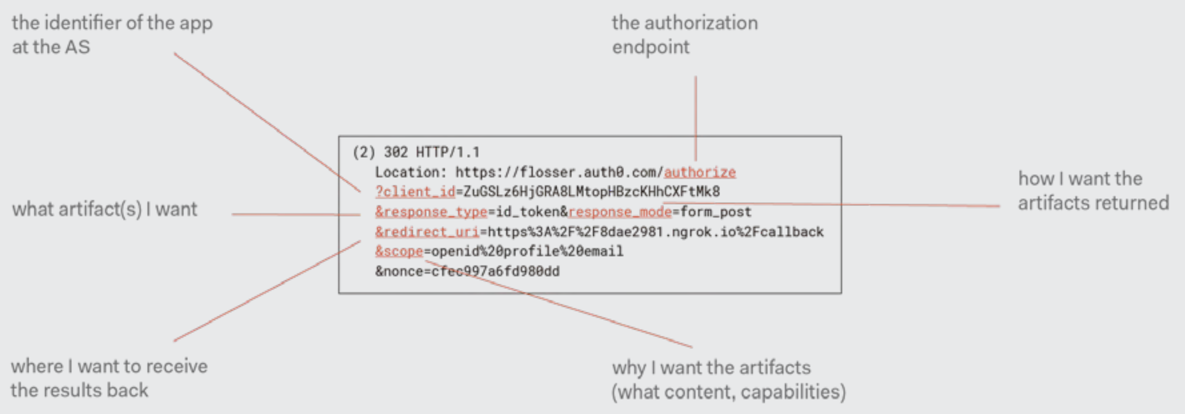
You might notice that in this authorization server, I'm showing only the authorization and discovery endpoints. I don't show the token endpoint because we don't use it in this particular flow.

The idea is that, as soon as this web application comes alive, the middleware will reach out to the discovery endpoint and learn everything it needs about the authorization server. In particular, it will get the authorization endpoint's address and the key to be used for checking signatures. We'll show how all those steps occur in detail later (see Metadata and Discovery section). For now, we'll focus on the authentication phase properly.

Let's see how the access plays out by describing each numbered step.

1. **Request Protected Route on Web App**
   In the first step, the browser reaches out to the application to get one particular route, which happens to be protected and hence not accessible by anonymous requests.

2. **Authorization Request Redirect**
   The middleware intercepts this call and emits an authorization request for the authorization server in response. The HTTP response has an HTTP 302 status code, i.e., it's a redirect. It has several parameters meant to communicate all the information necessary to perform the required authentication operation to the authorization server.


the identifier of the app at the AS
the authorization endpoint
what artifact(s) I want
how I want the artifacts returned
where I want to receive the results back
why I want the artifacts (what content, capabilities)

```
(2) 302 HTTP/1.1
   Location: https://flosser.auth0.com/authorize
   ?client_id=ZuGSLz6HjGRA8LMtopHBzcKHhCXFtMk8
   &response_type=id_token&response_mode=form_post
   &redirect_uri=https%3A%2F%2F8dae2981.ngrok.io%2Fcallback
   &scope=openid%20profile%20email
   &nonce=cfec997a6fd980dd
```

Figure 3.2

It’s really important to understand the anatomy of this message since all the other messages we'll see will be derivatives of this. Here, we're going to touch on all the most relevant parameters.

- **Authorization endpoint**
  The first element is the authorization endpoint. That's the address where we expect the authorization endpoint functionality to be for the authorization server.
- **Client ID**
  This *client_id* parameter is the identifier of your application at the authorization server. The authorization server has a bundle of configuration settings associated with your app, and it will bring those up in focus when it receives this particular client ID.
- **Response type**
  The *response_type* parameter indicates the artifact that I want. In this particular case, I want to sign in, so I need an ID token. Consequently, the value of the *response_type* parameter will be *id_token*. I can ask for a wide variety of artifacts, including combinations of artifacts; we'll see those combinations in detail.
- **Response mode**
  Response mode is how I want these artifacts to be returned to me. I have all the choices that HTTP affords me. I can get things in the query string, but this is usually a bad idea because artifacts end up in the browser history. I can get the artifacts in a fragment, which is still part of the URL but not transmitted to a server. I can get them as a form post (*form_post*), which is what we are using here. In this case, we just want to make sure that we post the token to our client. This way, we don't place stuff in the query string, which, as mentioned, is generally a bad practice from the security perspective. The use of a POST also allows us to have large tokens. In fact, if you place stuff anywhere but in a form post, you might run into size limitations.
- **Redirect URI**
  The *redirect_uri* parameter has a very important role. It represents the address in my application where I expect tokens and artifacts to be returned. I need to specify this because the tokens we use in this context are what we call bearer tokens. Bearer tokens are tokens that can be used just by owning them. In other words, I can use them directly without needing to do anything else, as other types of tokens might require. For example, other types of tokens may require me also to know a key and use it at the same time. But bearer tokens don't. You will hear much more about bearer tokens in the

token validation section (see Principles of Token Validation). So, it is imperative that I use only HTTPS so that no intermediary can interject itself and intercept traffic.
Also, it is very important that I specify the exact address I want the response to be sent back to. If I don't and, for example, instead of doing a strict match with the provided address, I allow callers to attach further parameters, I put communication security at risk. What might happen - and it did happen in the past - is that there might be flaws in the development stack I'm using that will cause my request to be redirected elsewhere. That would mean shipping my bearer tokens to malicious actors, and that's all they'd need to impersonate me. OAuth 2.0 and OpenID Connect are strict about this: the redirect URI you specify in the request must be an exact match of what you want.

- **Scope**
  The scope parameter represents the reason I'm asking for the artifacts. In the example above, I specified *openid, profile*, and *email*, which are scopes that cause the authorization server to issue an ID token with a particular layout. It's somewhat redundant with the earlier response type, but I'm also asking for enriching this ID token with the user's profile and email information, if present.
  In short, with the scope, I am specifying the reason I want the artifacts I am requesting. We will see that, when we use APIs, we ask for particular delegated permissions we want to acquire.

- **Nonce**
  The *nonce* parameter is mostly a trick for preventing token injection. At request time, I generate a unique identifier and save it somewhere (like in a cookie). This identifier is sent to the authorization server, and eventually, the ID token that I receive back will have a claim containing the same identifier. At that point, I'll be able to compare that claim with the identifier I saved, and I'll be confident that the token I received is the one I requested. If I receive a token with a different (or no) identifier, I have to conclude that the response has been forged and the token injected.

It is worth mentioning that I specified *form_post* as the value for *response_mode* because the default response mode of the ID token would be different (it would have been *fragment*); hence I had to override it explicitly. The following table shows the default response mode for each response type defined by OAuth 2.0 and OpenID Connect. If I omit *response_mode* in the request, the authorization server will apply its default value:

response_type	**default *response_mode***
code	query
token	fragment
id_token	fragment (query disallowed)
none	query
code token	fragment (query disallowed)
code id_token	fragment (query disallowed)
id_token token	fragment (query disallowed)
code id_token token	fragment (query disallowed)

Note that the upcoming OAuth 2.1 specification will deprecate the *token* response type, so all the response types containing *token* will be deprecated as well.

3. **Authorization Request**
   The next step for the browser is to honor the 302 redirection and actually perform a GET, hitting the authorization endpoint with all the parameters I just described.
   From now on, the authorization server does whatever it deems necessary to authenticate a user and prompt for consent. How this occurs isn't specified by OAuth 2.0 or OpenID Connect. The mechanics of user authentication, credentials gathering, and the like are a completely private matter of the authorization server as long as the eventual response is in the standard's format. You can have multi-factor authentication, multiple pages, or one single page. It doesn't matter as long as you come out with a standard result.

4. **Authorization Response**
   Once everything works out, you get an HTTP response with a 200 status code. This means that you have successfully authenticated with the authorization server. The authorization server will set a cookie that represents your session with it. So, if you need to hit the authorization endpoint again later on, you will not have to enter credentials to sign in explicitly. You might have to give more consent, for example, but you shouldn't have to re-enter credentials.
   The other important part to note here is the ID token we requested. It is being returned as a parameter in the form post we are getting. You can see in the body of the HTML being returned that the JavaScript **onload** event is wired up to submit a form automatically.

5. **Send the Token to the Application**
   As soon as the page returned by the authorization server gets rendered, it will post the form to our application. This means that the requested ID token has finally been sent to my web application.

6. **Token Validation and Web App Session Creation**
   What happens now is pretty much the same thing that we studied earlier in the web sign-on scenario in the first chapter, Introduction to Digital Identity. The application receives the ID token and decides whether or not it likes it according to all the various trust rules and what it has learned from the discovery endpoint. If it likes it, the app will emit an HTTP 302 response with its own cookie. Thanks to that cookie representing an authenticated session with my app, I will not need to get the ID token again as long as the cookie is valid. With the cookie creation, the app emits an HTTP 302 response, redirecting the browser to the original route it requested.

7. **Request Protected Route with Authorization**
   As the browser honors the redirect, we end up where we started: we request a protected route, but this time, we present a session cookie with it.
   If you compare the original request with this redirect, you will discover that it is exactly the same request but with a cookie coming along.

8. **Access the Protected Route**
   Finally, after this long back-and-forth, we can get our response: an HTTP 200 response with a page in the body.
   From now on, every subsequent request to the application will carry the session cookie, proving that an authenticated session is in place.

Anatomy of an ID Token

As we said earlier, the ID token is an artifact that proves that successful authentication occurred. We have two ways of requesting it: using a *response_type* parameter with the *id_token* value and using a *scope* parameter with the *openid* value.

The reason we have two mechanisms is that the authors of the specifications wanted to be able to use OpenID Connect even if your SDK was only based on OAuth 2.0. In fact, at the OAuth 2.0 time, there was no ID token in the enumeration of a response type. Since scopes are completely generic as a parameter, then the ability to use one particular scope that would cause the authorization server to return an ID token was a great way of being backward-compatible. Today, it's a great way of getting confused, but now that you know, you no longer run this risk.

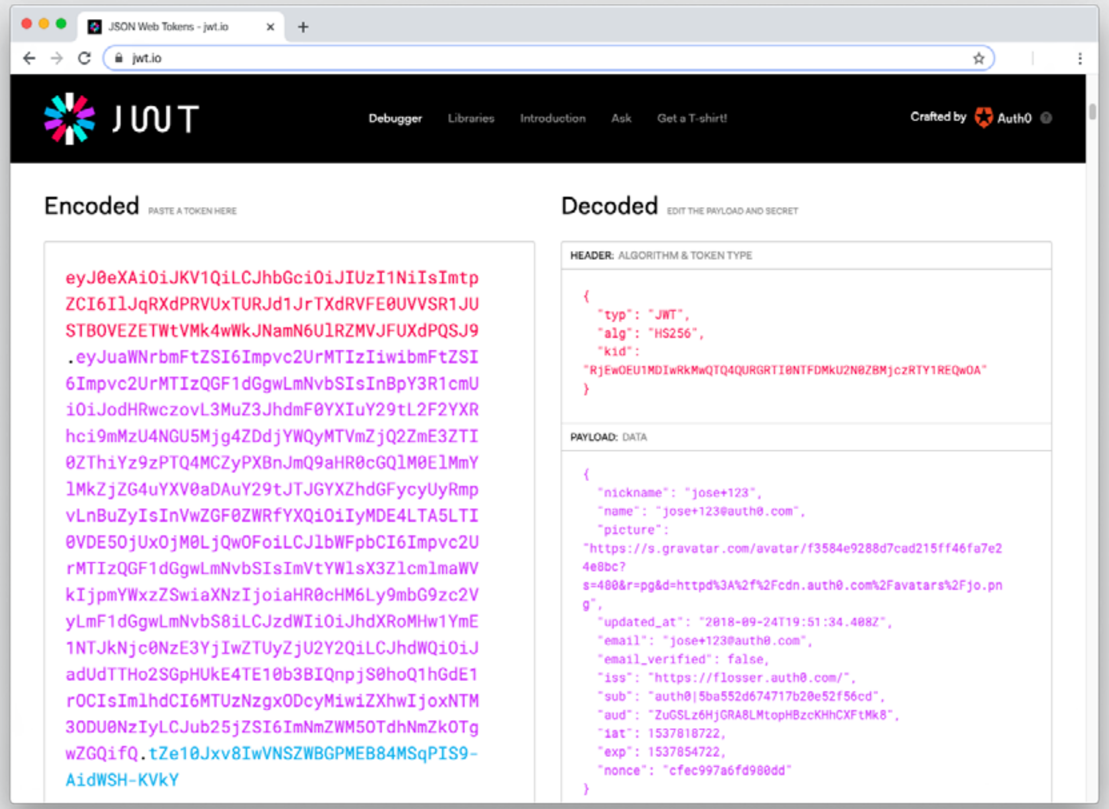
OpenID Connect defines the ID token as a fixed format, the JSON Web Token (JWT) format. The specification actually defines not just the format but also the list of claims that must be present in an ID token. In addition, it even tells you in normative terms what you need to do in order to validate some of those claims. As we said, if I include a *profile* or *email* value in the scopes of my request, I will cause the content of the ID token to look different.

Just to get a feeling of it, here you can see what you would normally see on the wire:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6IlJqRXdPRVUxTURJd1JrTXdRVFE0UVVSR1JUSTBOVEZETWtVMk4wWkJNamN6UlRZMVJFUXdPQSJ9
.eyJuaWNrbmFtZSI6Impvc2UrMTIzIiwibmFtZSI6Impvc2UrMTIzQGF1dGgwLmNvbSIsInBpY3R1cmUiOiJodHRwczovL3MuZ3JhdmF0YXIuY29tL2F2YXR
hci9mMzU4NGU5Mjg4ZDdjYWQyMTVmZjQ2ZmE3ZTI0ZThiYz9zPTQ4MCZyPXBnJmQ9aHR0cHMlM0ElMkYlMkZjZG4uYXV0aDAuY29tJTJGYXZhdGFycyUyRmp
vLnBuZyIsInVwZGF0ZWRfYXQiOiIyMDE4LTA5LTI0VDE5OjUxOjM0LjQwOFoiLCJlbWFpbCI6Impvc2UrMTIzQGF1dGgwLmNvbSIsImVtYWlsX3ZlcmlmaWV
kIjpmYWxzZSwiaXNzIjoiaHR0cHM6Ly9mbG9zc2VyLmF1dGgwLmNvbS8iLCJzdWIiOiJhdXRoMHw1YmE1NTJkNjc0NzE3YjIwZTUyZjU2Y2QiLCJhdWQiOiJ
adUdTTHo2SGpHUkE4TE10b3BIQnpjS0hoQ1hGdE1rOCIsImlhdCI6MTUzNzgxODcyMiwiZXhwIjoxNTM3ODU0NzIyLCJub25jZSI6ImNmZWM5OTdhNmZkOTg
wZGQifQ. ngwlT3mvl3OrvlYYZ02xYhhE-
QGTtf3M_gsJGPtKG5kqhEFWvWHstzWMJE6ZjAapHYitKQg0zDpEbqixaB6PEqz7b09SwLeotZSUBGnfQHU5gC8cpXfMPlTkhyEcERz9yOxO3TbVJDGfoEJZ
eu
_To7BGA3vlvEn7XWHD0Oz45Ht2xtklISfW4vXno_ahZMbLufOJFkpJqtUvHMmd9hVy33uZXp_Z7Vggfk_LDD58XKaJJ8Z9WhPUr1RFl14IPTNEmtmgSEWXz
ds6GYA-Ap5OH2NWIKZe59eDgqi64GPhhjK0u8qSUAue6oQa7M_yw817sJA9yKHdg5mZl4piTCA
```

Figure 3.3

That's what a JWT token normally looks like, with its Base64 encoded components. If you go to jwt.io, which is a very handy utility offered by Auth0, you can actually paste the bits of your ID token and see it automatically decoded. The following picture shows an example of such decoding:



Encoded PASTE A TOKEN HERE

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiIsImtp
ZCI6IlJqRXdPRVUxTURJd1JrTXdRVFE0UVVSR1JU
STBOVEZETWtVMk4wWkJNamN6UlRZMVJFUXdPQSJ9
.eyJuaWNrbmFtZSI6Impvc2UrMTIzIiwibmFtZSI
6Impvc2UrMTIzQGF1dGgwLmNvbSIsInBpY3R1cmU
iOiJodHRwczovL3MuZ3JhdmF0YXIuY29tL2F2YXR
hci9mMzU4NGU5Mjg4ZDdjYWQyMTVmZjQ2ZmE3ZTI
0ZThiYz9zPTQ4MCZyPXBnJmQ9aHR0cGQlM0ElMmY
lMkZjZG4uYXV0aDAuY29tJTJGYXZhdGFycyUyRmp
vLnBuZyIsInVwZGF0ZWRfYXQiOiIyMDE4LTA5LTI
0VDE5OjUxOjM0LjQwOFoiLCJlbWFpbCI6Impvc2U
rMTIzQGF1dGgwLmNvbSIsImVtYWlsX3ZlcmlmaWV
kIjpmYWxzZSwiaXNzIjoiaHR0cHM6Ly9mbG9zc2V
yLmF1dGgwLmNvbS8iLCJzdWIiOiJhdXRoMHw1YmE
1NTJkNjc0NzE3YjIwZTUyZjU2Y2QiLCJhdWQiOiJ
adUdTTHo2SGpHUkE4TE10b3BIQnpjS0hoQ1hGdE1
rOCIsImlhdCI6MTUzNzgxODcyMiwiZXhwIjoxNTM
3ODU0NzIyLCJub25jZSI6ImNmZWM5OTdhNmZkOTg
wZGQifQ.tZe10Jxv8IwVNSZWBGPMEB84MSqPIS9-
AidWSH-KVkY
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "HS256",
  "kid":
"RjEwOEU1MDIwRkMwQTQ4QURGRTI0NTFDMkU2N0ZBMjczRTY1REQwOA"
}
```

PAYLOAD: DATA

```
{
  "nickname": "jose+123",
  "name": "jose+123@auth0.com",
  "picture":
"https://s.gravatar.com/avatar/f3584e9288d7cad215ff46fa7e2
4e8bc?
s=480&r=pg&d=httpd%3A%2f%2Fcdn.auth0.com%2Favatars%2Fjo.pn
g",
  "updated_at": "2018-09-24T19:51:34.408Z",
  "email": "jose+123@auth0.com",
  "email_verified": false,
  "iss": "https://flosser.auth0.com/",
  "sub": "auth0|5ba552d674717b20e52f56cd",
  "aud": "ZuGSLz6HjGRA8LMtopHBzcKHhCXFtMk8",
  "iat": 1537818722,
  "exp": 1537854722,
  "nonce": "cfec997a6fd980dd"
}
```

Figure 3.4

On the right side, you can see a **header** that describes the shape of this specific JWT. In particular, by examining the header content, we find that this token is in JWT format, what algorithm has been used for signing it, and a reference to the key required to validate the signature, which in this case corresponds to the key that we downloaded from the discovery endpoint (more on that in a moment).

If you look at the payload, you'll find that it contains the actual information we expected to retrieve. Going into more details, we have:

- The **issuer** (*iss*), which is a string representing the source of the token. It is the entity behind the authorization server - like the key, also found via the discovery endpoint.
- The **audience** (*aud*), which represents the particular application the token has been issued for. It is very important to check this claim. As an app receives this token, the middleware used for validating it will compare what was configured to be the app identifier (in the case of sign-in and ID tokens, that will correspond to the client ID of the app) with the audience claim. If there is a mismatch, that means that someone stole a token from somewhere else and is trying to trick the app into accepting it.
- The **issued-at** (*iat*) and **expiration** (*exp*) are coordinates used to evaluate whether this token is still within its validity window or if, being expired, it can no longer be accepted. During the API discussion, we'll see that access tokens and ID tokens typically have a limited validity time.
- All the other claims are pretty much identity information about the user, which is present in the ID token only because I asked for *profile* and *email* in the *scope* parameter.

Principles of Token Validation

We've been talking about validating tokens quite a lot, relying on the intuition that it entails validating signatures and performing metadata discovery. Let's explore the matter in more detail and have a more organic discussion about what it means to validate tokens.

We have seen the function that tokens perform in a couple of scenarios. We have seen signing in with SAML. We have seen access tokens for calling APIs, and in particular, right now, we have seen how to use an ID token to sign in. All those scenarios entail an entity, the resource, receiving a token and making a decision about whether it entitles the caller to perform whatever operation the caller is attempting. How does the resource make that decision?

Subject Confirmation

Subject confirmation is a concept we inherit from SAML. In particular, the subject confirmation method determines how a resource decides whether a token has been used correctly.

- **Bearer**
  Is the simplest. It is similar to finding 20 dollars on the floor. You pick up the money, go wherever you want to use this money, use it, and you're going to get the goods or services you are paying for. No further questions will be asked because all it takes to use 20 dollars is to *own* those 20 dollars and for them to change hands. That's the substance of the *bearer* subject confirmation method. If you have the bits of a token in your possession, you are entitled to use the token

- **Proof of possession**
  Is something more advanced. In *proof of possession*, you have a token containing a key of some kind in some encrypted section. This encryption is specifically done for the intended recipient of the token. The idea is that when a client obtains such a token, they also receive a separate session key, the same key embedded in the encrypted section of the token. When the client sends a message to the intended recipient, it attaches the token as in the *bearer* case, but it also uses this session key to do something - like signing part of a message.

> When the resource receives the token and the message, they will validate the token in the usual way as we described for the *bearer* method. Once that is done, they will extract the session key from the portion that was encrypted for them. They'll use the session key to validate the signature in the message. If the validation works, the recipient will know for certain that the caller is the original requestor that obtained the token in the first place. Otherwise, they would not have been able to use the session key.
> This mechanism is more secure than the *bearer*: an attacker intercepting the message would be able to replay the token, but without knowledge of the session key, they could not perform the additional signature and provide *proof of possession*.

Until recently, almost no one used *proof of possession* in OAuth 2.0 or OpenID Connect. But *proof of possession* is now coming back. A recent specification, *Demonstrating Proof of Possession* (DPoP), shows how to use the mechanism I just described in OAuth 2.0 and OpenID Connect, although it will take some time before it is widely adopted. So, to all intents and purposes, you can think of Bearer tokens as being the law of the land. There is another concept - the *sender constraint* - but I'll talk more about it when we deal with native clients (Chapter 5, Desktop and Mobile Apps).

Format Driven Validation Checks

In OAuth 2.0, access tokens have no format. The standard doesn't specify any format, mostly because originally, it was intended for a scenario where the authorization server and the resource server are co-located and can share memory.

Think, for example, of the scenario we described in the first chapter, where Gmail is the resource server with its own APIs, and it's also the authorization server.

In that particular scenario, those two entities can share memory. They can have, for example, a shared database. So, when a client asks for an access token, this access token can be just an opaque string that is the primary key in a specific table where the authorization server has saved the consent granted by the user to the client.

When the client makes a call to the resource server presenting this token, the resource server grabs the token and just uses it to find the correct row in the database and then the consented permissions. The resource server uses that information to make an authorization decision.

This scenario complies with the spirit of the spec - and also the letter of the spec - and we didn't need to mandate any specific format.

However, in the case of OpenID Connect, we did define a format for the ID token. We expected the receiver actually to look inside a token and perform validation steps. This typically happens when the resource and authorization servers are not co-located and, hence, cannot use shared memory to communicate. In those cases, you typically (but not always) rely on an agreed-upon format.

Also, in the SAML case, we defined a format, a set of instructions on how to encode a token.

In the case of format-driven validation checks, there are certain constraints that apply pretty much to every format, and in particular, to JWT:

- **Signature for integrity**
  Your token is signed, and we have seen the reasons we want to sign a token: to be sure of the token's origin and to prevent tampering in transit. The token must provide some indication about the key and the algorithm used so its recipient can check its signature.

- **Infrastructural claims**
  Token formats typically include infrastructural claims, which provide information the token recipient must validate to determine whether the incoming token should be accepted. One notable example of those claim types is the *issuer*, which is, to say, the identifier of the entity that issued (and signed) the token, and that should correspond to one of the issuers trusted by the intended recipient. Another common infrastructural claim, the *audience*, says for whom a token is meant to. You need the *audience claim* to have a way of validating that the token is actually for a specific recipient. You also need *expiration times* claims: tokens have typically restricted validity so that there is the opportunity to revoke them.
  Those are all claims that you would expect tokens to have and that the middleware is typically on point to validate.

Alternative Validation Strategy: Introspection

There is a different way of validating tokens, which goes under the name of **introspection**. With this approach, the resource receiving a token considers it opaque. It may happen because it doesn't have the capability to validate the token. It should be rare in the JWT case because checking a JWT is pretty trivial and can be done in any dev stack.

However, imagine that, for some reason, you cannot assume that incoming tokens are in a format that you know how to validate. You can take the incoming token and send it to the **introspection endpoint**, which is an additional endpoint that authorization servers can expose. Given that you connect to the introspection endpoint using HTTPS, you can actually validate the identity of the server itself. You can be confident that you are sending the token where it's meant to go, as opposed to a malicious site. The authorization server examines the token, determines whether that token is valid or not, and, if it is valid, sends down the same channel the content of the token itself (e.g., claims).

In a nutshell, the resource server sends back tokens to the authorization server saying, "Please tell me whether it's valid or not." The authorization server can render a decision and send it back to the client, along with the content of the token, so that the resource can peek inside.

Personally, I'm not crazy about introspection, mostly because it's brittle. You need to have the authorization server up and available, and if your application is very chatty, you might get throttled, for example. Also, with this approach, you need to wait until you have one extra network round trip before you can actually make an access control decision about the resource you're calling. You might run out of outgoing HTTP connections, which typically live in a pool. It's a lot of work.

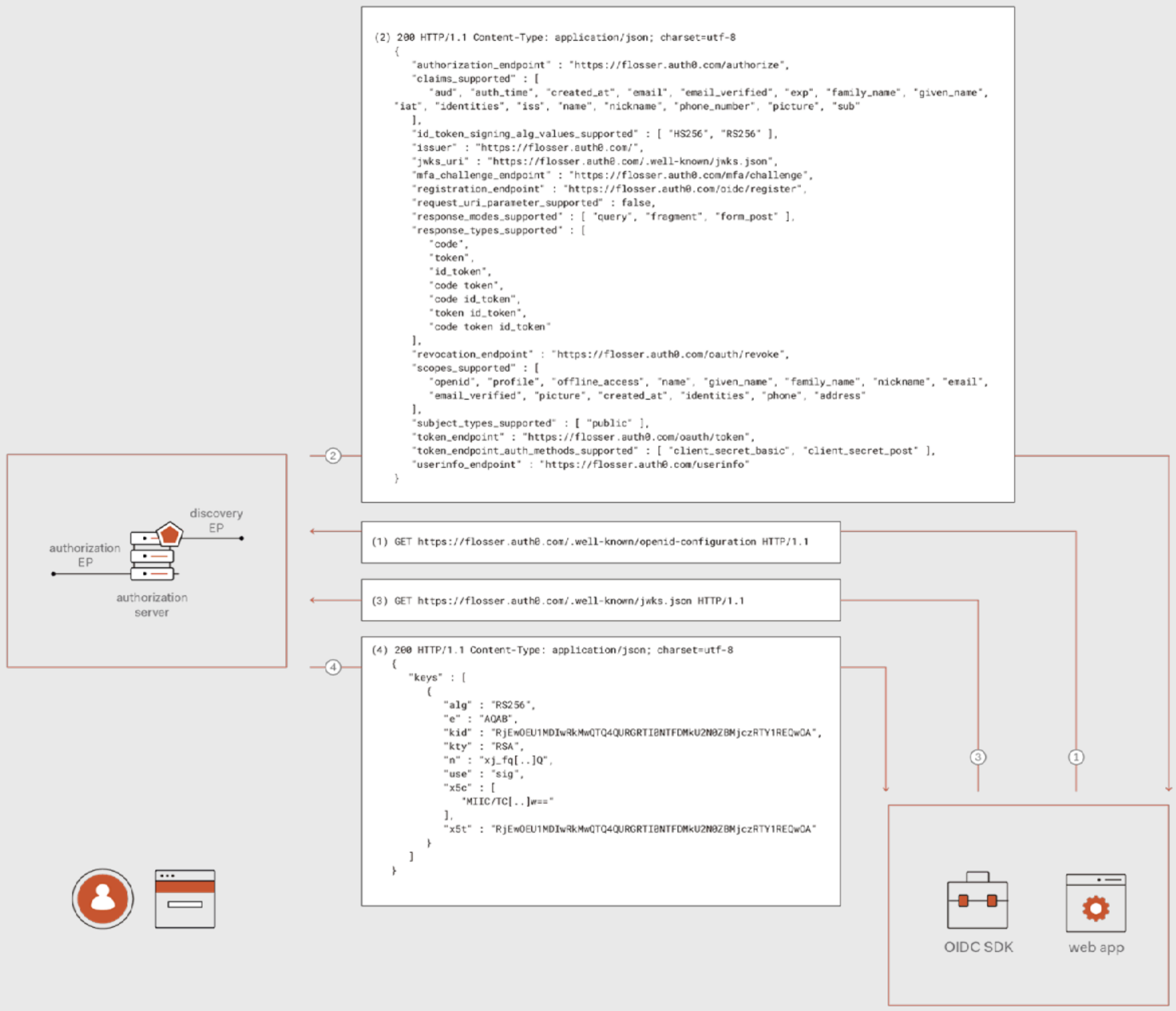
Sometimes there are no alternatives. But in general, for Auth0, given that we always use JWTs and public cryptography, it's usually better if you validate your own token at your API.

Metadata and Discovery

Token validation middleware discovers the values expected in valid tokens through the *discovery endpoint*. The middleware simply hits the URL *./well-known/openid-configuration*, which is defined by OpenID Connect, and retrieves validation information according to the specification.

The document published at this URL typically contains direct information that we need to have, like the issuer value, the addresses of our authorization endpoint, and similar. It also connects to a different file containing the actual keys, which could be literally the bits of X.509 public key certificates.

Let’s take a look at how middleware extracts validation information from the discovery endpoint by following the numbered steps in Figure 3.5.



```
(1) GET https://flosser.auth0.com/.well-known/openid-configuration HTTP/1.1
```

```
(2) 200 HTTP/1.1 Content-Type: application/json; charset=utf-8
   {
      "authorization_endpoint" : "https://flosser.auth0.com/authorize",
      "claims_supported" : [
         "aud", "auth_time", "created_at", "email", "email_verified", "exp", "family_name", "given_name",
   "iat", "identities", "iss", "name", "nickname", "phone_number", "picture", "sub"
      ],
      "id_token_signing_alg_values_supported" : [ "HS256", "RS256" ],
      "issuer" : "https://flosser.auth0.com/",
      "jwks_uri" : "https://flosser.auth0.com/.well-known/jwks.json",
      "mfa_challenge_endpoint" : "https://flosser.auth0.com/mfa/challenge",
      "registration_endpoint" : "https://flosser.auth0.com/oidc/register",
      "request_uri_parameter_supported" : false,
      "response_modes_supported" : [ "query", "fragment", "form_post" ],
      "response_types_supported" : [
         "code",
         "token",
         "id_token",
         "code token",
         "code id_token",
         "token id_token",
         "code token id_token"
      ],
      "revocation_endpoint" : "https://flosser.auth0.com/oauth/revoke",
      "scopes_supported" : [
         "openid", "profile", "offline_access", "name", "given_name", "family_name", "nickname", "email",
         "email_verified", "picture", "created_at", "identities", "phone", "address"
      ],
      "subject_types_supported" : [ "public" ],
      "token_endpoint" : "https://flosser.auth0.com/oauth/token",
      "token_endpoint_auth_methods_supported" : [ "client_secret_basic", "client_secret_post" ],
      "userinfo_endpoint" : "https://flosser.auth0.com/userinfo"
   }
```

```
(3) GET https://flosser.auth0.com/.well-known/jwks.json HTTP/1.1
```

```
(4) 200 HTTP/1.1 Content-Type: application/json; charset=utf-8
   {
      "keys" : [
         {
            "alg" : "RS256",
            "e" : "AQAB",
            "kid" : "RjEw0EU1MDIwRkMwQTQ4QURGRTI0NTFDMkU2N0ZBMjczRTY1REQwOA",
            "kty" : "RSA",
            "n" : "xj_fq[..]Q",
            "use" : "sig",
            "x5c" : [
               "MIIC/TC[..]w=="
            ],
            "x5t" : "RjEw0EU1MDIwRkMwQTQ4QURGRTI0NTFDMkU2N0ZBMjczRTY1REQwOA"
         }
      ]
   }
```

Figure 3.5

1. **Request Configuration**
   At load time or even the first time you receive a message, the middleware contacts the discovery endpoint.
   That's a simple matter of making an HTTP GET request to the *./well-known/openid-configuration* endpoint of the authorization server.

2. **Receive Configuration Document**
   What you get back is a big JSON document with all the values required to validate incoming tokens.
   For example, just to highlight some of these values, you have the address of the authorization endpoint (*authorization_endpoint*), the value of the issuer (*issuer*), which is the value that we are supposed to validate against, a list of supported claims (*claims_supported*), the supported response modes (*response_modes_supported*), and a pointer to the file where all keys are kept (*jwks_uri*).

3. **Request Keys**
   The next step is to actually make a GET request to the address where the keys are published.

4. **Receive Keys**
   The result of that request will be another file containing a collection of keys with their respective supported algorithm (*alg*), their identifier (*kid*), and the bits of the public key. The middleware programmatically downloads all of that stuff and keeps it ready.
   Those keys will occasionally roll because it's good practice to change them. Your middleware will simply have to reach out and re-download these keys when it happens.

Chapter 4

Calling an API from a Web App

In this chapter, we move our attention to calling APIs. This is the quintessential scenario addressed by OAuth 2.0: delegated access to APIs is the main reason for OAuth's existence.

Most of the discussion will focus on the canonical grant OAuth 2.0 offers to address the delegated API access scenario, the Authorization *Code grant*. We'll also take a look at other grants, such as the *Hybrid* flow and the *Client Credentials* grant, which can be used to call APIs in slightly different scenarios.

The Authorization Code Grant

At a high level, the way we typically invoke an API from a web application is roughly the same way we'd call an API from any client flavor. Details will differ, as we will see throughout the book.

Depending on the client's flavor, we'll use different grants with different properties. In particular, in this chapter, we want to focus on the scenarios in which a web application calls an API from its server-side code. For that purpose, we use the OAuth 2.0 **Authorization Code grant**. The Authorization Code grant, **Code grant** from now on for brevity, empowers one web application to access an API on *behalf of a user* and *within the boundaries of what the user granted consent for*. This is the grant we encountered when introducing OAuth 2.0 in Chapter 1.

In the section *Layering Sign In on Top of OAuth 2.0: OpenID Connect* of Chapter 1, we've seen that some people tried to stretch this grant to achieve sign-in, as opposed to invoking an API. In the same section, we have seen how if you just use this grant to obtain and use access tokens for signing in, things don't work out that well. We have seen how OpenID Connect is layered on top of this grant to achieve sign-in the right way, and we'll have more considerations about it in this chapter. At this point, I just want to stress that what we are looking at in this chapter is aimed at calling APIs and not at signing in.

Another important concept to grok upfront is that the Code grant will only empower an application to do up to as much as the user can already do and no more. If anything, the application will usually end up having fewer access rights. Users cannot use the Code grant to grant applications access to the resources the users themselves don't own or have the rights to. When thinking about OAuth 2.0 and the Code grant in particular, it's easy for people to get confused. They observe that APIs grant access to a call depending on the presence of scopes in the token. That lends to the belief that the scopes themselves are what grant the client the privileges to access the resource. Actually, the scopes select what privileges the user already has and is delegating to the client.

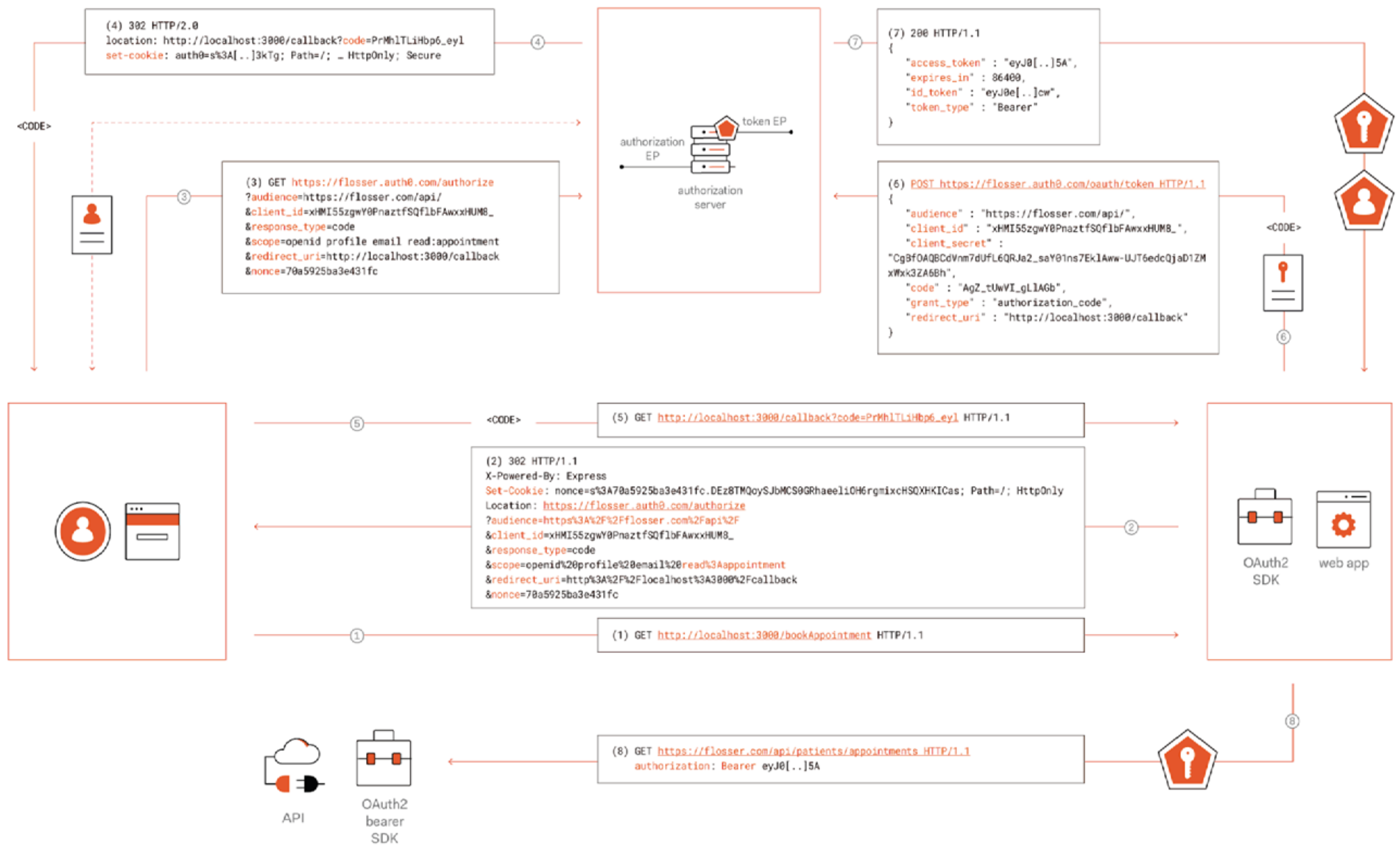
I just want to stress that **the Authorization Code grant is a delegated flow**. It allows clients to do things on the user’s behalf, which means that the user’s capabilities are a hard limit for what an application can do on the user’s behalf. In other words, a client obtaining a token via Code grant cannot do more than the user can. If you need a client to do more than the user can do, which is a common scenario, then you need to switch to a different flow in which permissions are granted directly to the application that needs it, with no user involvement. Clear as mud? Don’t worry. We'll revisit those points later in the chapter.

In the last chapter, we explored how to perform web sign-in through the front channel, which afforded us the luxury of implementing the full scenario without any secrets. As you witnessed in the detailed descriptions of flows and network traces, no secret came into play. In the *Authorization Code* grant, however, using an application credential such as a client secret is inevitable. Whenever the web app redeems an authorization code, it needs to authenticate as a client to the authorization server.

We will approach the delegated API invocation scenario differently depending on whether one needs to access the APIs only while a user is present and currently signed in to the application, or whether one needs to acquire permanent access to the APIs and perform calls to these APIs even when no user is present.

My favorite example is an application that can publish tweets at an arbitrary time. Personally, I don't like to wake up early in the morning; I really hate it. Nonetheless, it turns out that tweets get the best exposure when they come out pretty early. The fact that I'm based on the West Coast makes things even worse: if I have to publish tweets manually at a time that should be considered morning in the entire North America, I'd have to wake up really early. Luckily, there are applications I can use for tweeting on my behalf at whatever time I schedule beforehand. Those applications are a typical example of a client needing an access token always available to call the Twitter API on my behalf, regardless of whether I am currently signed in an active session or am blissfully still asleep. This is one of the classic scenarios, offline access, demonstrating the need and intended usage of a very important artifact - the refresh token. Once again, we'll explore this scenario in detail in this chapter.

Without further ado, let's dive into the details of the Authorization Code grant with the help of the diagram in Figure 4.1.



```
(1) GET http://localhost:3000/bookAppointment HTTP/1.1
```

```
(2) 302 HTTP/1.1
X-Powered-By: Express
Set-Cookie: nonce=s%3A70a5925ba3e431fc.DEz8TMQoySJbMCS0GRhaeeliOH6rgmixcHSQXHKICas; Path=/; HttpOnly
Location: https://flosser.auth0.com/authorize
?audience=https%3A%2F%2Fflosser.com%2Fapi%2F
&client_id=xHMI55zgwY0PnaztfSQflbFAwxxHUM8_
&response_type=code
&scope=openid%20profile%20email%20read%3Aappointment
&redirect_uri=http%3A%2F%2Flocalhost%3A3000%2Fcallback
&nonce=70a5925ba3e431fc
```

```
(3) GET https://flosser.auth0.com/authorize
?audience=https://flosser.com/api/
&client_id=xHMI55zgwY0PnaztfSQflbFAwxxHUM8_
&response_type=code
&scope=openid profile email read:appointment
&redirect_uri=http://localhost:3000/callback
&nonce=70a5925ba3e431fc
```

```
(4) 302 HTTP/2.0
location: http://localhost:3000/callback?code=PrMhlTLiHbp6_eyl
set-cookie: auth0=s%3A[..]3kTg; Path=/; _ HttpOnly; Secure
```

```
(5) GET http://localhost:3000/callback?code=PrMhlTLiHbp6_eyl HTTP/1.1
```

```
(6) POST https://flosser.auth0.com/oauth/token HTTP/1.1
{
   "audience" : "https://flosser.com/api/",
   "client_id" : "xHMI55zgwY0PnaztfSQflbFAwxxHUM8_",
   "client_secret" :
"CgBfOAQBCdVnm7dUfL6QRJa2_saY01ns7EklAww-UJT6edcQjaD1ZM
xWxk3ZA6Bh",
   "code" : "AgZ_tUwVI_gLlAGb",
   "grant_type" : "authorization_code",
   "redirect_uri" : "http://localhost:3000/callback"
}
```

```
(7) 200 HTTP/1.1
{
   "access_token" : "eyJ0[..]5A",
   "expires_in" : 86400,
   "id_token" : "eyJ0e[..]cw",
   "token_type" : "Bearer"
}
```

```
(8) GET https://flosser.com/api/patients/appointments HTTP/1.1
    authorization: Bearer eyJ0[..]5A
```

Figure 4.1

The diagram depicts the usual actors we encountered in Chapter 2:

- On the far left, the user and their browser.
- The authorization server, on top. Note that this time, both the authorization and the token endpoints are present in the picture, as both will come into play.
- A web application roughly in the middle.
- The API the web app needs to call as part of our scenario.

Just like we did during the first explanation of the OAuth 2.0 flow in Chapter 1, section Delegated Authorization: OAuth 2.0, we assume the user has already signed in to the web application. We don't know how that sign-in operation occurred, and we don't care in this context - the API invocation operation can be performed independently of the sign-in (although we will later see, in the section on Hybrid flow, that there are potential synergies there). Let’s examine the message sequence in detail.

1. **Route Request**
   In our sample scenario, the user hits a web application route that allows the user to book an appointment. Booking an appointment requires accessing the booking API on behalf of the user, which causes the web app to generate a request for delegated access. Note, if you compare the equivalent step in the flow described in Chapter 3, section *The Implicit Grant with Form Post* for the sign-in operation, you will notice that the web app does not have a middleware in front to intercept the route request. In this case, the route isn’t the asset we want to protect: requesting that route just happens to be the thing that triggers the need to acquire a token to call an API. The logic necessary to generate the associated delegated authorization request is, in fact, inside the app codebase itself (although it will often be implemented by an SDK rather than from scratch).

2. **Authorization Request**
   The application’s reaction to the request is somewhat familiar: a 302 HTTP status code response with a message for the authorization server. However, you can see a number of differences with the equivalent step 2 in section The Implicit Grant with form_post of Chapter 3.

First, we are setting a cookie to track the *nonce* value (see Chapter 3, section Authorization Request Redirect for more details), as besides the access token needed for accessing the API, we'll also be asking for an ID token. The ID token is useful in this flow, knowing a bit more about the transaction, given that the access token itself is opaque to the client. More details later in this chapter.

Next, in the captured trace message, we have the authorization endpoint. Let's ignore the *audience* parameter for a second. The next entry is the *client_id,* which represents the client ID identifying the web app at the authorization server.

The *response_type* for this particular grant is *code*. We want to obtain a code from the authorization endpoint, which the web app will later exchange via the token endpoint for an access token. We don't need to specify the response mode because we are okay with a default response mode, which in the case of code response type is *query* - meaning that we expect the authorization server to return the authorization code in a query string parameter.

Next, we find the *scope* parameter. This message includes all the same scope values encountered earlier - *openid, profile,* and *email* - indicating that we require an ID token alongside the code. This time, however, we aren't requesting an ID token for sign-in purposes; we just want to have some information about who the resource owner granting permission in this transaction is. Without an ID token, that is to say, something the client itself can consume, we would have no way to know. We'd just blindly get an access token and use it with no indication about the identity of the user who obtained it.

The scope collection includes a scope value we haven't encountered yet, *read:appointment*. That scope value represents a permission exposed by the API we want to invoke; in other words, one of the things that can be done when using that particular API and can be gated by an authorization check. By presenting that scope value in the authorization request, the client says to the authorization server, "This web application wants to exercise the *read:appointment* privilege on behalf of the user". That's something that the authorization server needs to know. It will determine important details in the way the request is handled, such as the content of the consent prompt presented to the user and the actual outcome of granting the delegated permissions.

The next parameter represents the redirect URI, which you are already familiar with. The last parameter in the captured message is the nonce, a token injection prevention mechanism we encountered earlier in the book.

Now that we covered every message parameter in detail, let's revisit the *audience* parameter. When requesting an access token for an API protected by Auth0, a client is required to specify one extra parameter, called **audience**, indicating the identity of the resource to which the client is requesting access.

The core OAuth 2.0 specification does not contain any parameter performing this function, mostly because there is an underlying assumption (though not a requirement) that the resource server and authorization server are co-located. This assumption makes it unnecessary to identify which resource server the request refers to. For a concrete example of this scenario, consider how Facebook uses OAuth 2.0 for gating access to its Graph API. The Facebook authorization server can only issue access tokens for the Facebook Graph API; there is no other resource server in the picture. The only latitude left to clients is to specify different scopes for that one resource server, the Facebook Graph. Different scopes will express different permissions and operations I intend to exercise, but they will all refer to the same resource server, which doesn't need to be explicitly named in the authorization request. Similar considerations hold for Google, Dropbox, and other popular services. Whenever clients get tokens from those services, they are always calling the provider's own APIs, whose identity results self-evident from the context without requiring an identifier in the request.

When the solution includes a third-party authorization server, like in the case of an Auth0 customer leveraging the Auth0 authorization server to secure its own custom API, the topology allows the same authorization server to gate access for a multitude of resources, which can all live in different places. In that scenario, the client needs to be able to specify which resource it intends to request access to.

There are multiple ways a message could be constructed to include explicit references to a particular resource server. For example, an API might embed a resource server identifier in individual scope strings themselves. However, this approach has issues: scope strings could get really long and hard to read. Also, including multiple scopes referring to different resources in the same request might generate ambiguity about which resources the resulting access token could be used with.

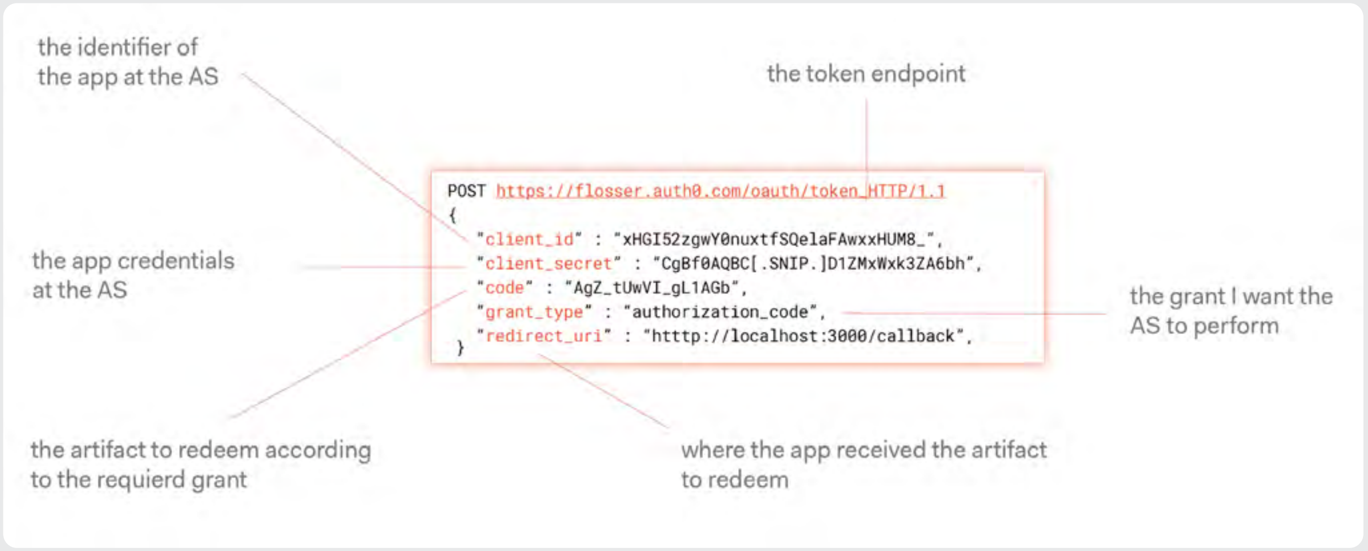
Given those complications, Auth0 and other identity vendors decided to introduce a dedicated parameter for identifying resources. Azure AD, for example, has a *resource* parameter whose semantics are equivalent to Auth0's *audience*.
Since those individual vendor decisions have been made, the IETF OAuth 2.0 working group officially recognized the usefulness of such primitives and issued a new specification, OAuth 2.0 Resource Indicators. This specification extends OAuth 2.0 with a *resource* parameter, which is, to all intent and purposes, equivalent to Auth0's audience. We plan to start accepting those standard parameters too in a future update.

3. **302 Redirect Execution**
   Next, the browser executes the 302 HTTP status code redirection by sending the message we examined toward the authorization endpoint.

4. **Authorization Response**
   Upon receiving the authorization request, the authorization server takes care of the interactive portion of the flow.
   The authorization endpoint decides what's necessary for authenticating the user, and goes through it. Then, it presents them with a consent prompt saying, "Hey, client X wants to read appointments on your behalf." When the user grants consent, the authorization endpoint returns its response with the requested authorization *code* in the query string, in accordance with the *response_type* we asked for. Also, the response includes the usual *set-cookie* command with which the authorization server records in the browser that an authentication session has been established.

5. **Providing the Authorization Code to the Web App**
   At this point, the browser simply executes the redirect that will dispatch the authorization code to the web application. From this moment on, the web application will continue the flow on the server side.

6. **Redeeming the Authorization Code**
   The web application combines the authorization code with its own client credentials and sends them in a message to the token endpoint.



```
POST https://flosser.auth0.com/oauth/token HTTP/1.1
{
   "client_id" : "xHGI52zgwY0nuxtfSQelaFAwxxHUM8_",
   "client_secret" : "CgBf0AQBC[.SNIP.]D1ZMxWxk3ZA6bh",
   "code" : "AgZ_tUwVI_gL1AGb",
   "grant_type" : "authorization_code",
   "redirect_uri" : "htttp://localhost:3000/callback",
 }
```

Figure 4.2

The message to the token endpoint is in the form of an HTTP POST request where the app presents its *client_id* and *client_secret*, the authorization *code* received from the front channel, and a new parameter, the **grant_type.** The message layout is shown, annotated, in Figure 4.2.

Every time an application talks to the token endpoint, it has to specify the desired grant type, letting the authorization server know how to interpret the request. In this particular case, the desired flow is the **authorization_code grant**. That tells the authorization server to search for an authorization code in the message, and to consider the client ID and secret in the context of this specific grant. If, for example, the request would have specified *client_ credentials* as the grant type, a flow we'll discuss later on, then the authorization server would have ignored the authorization code, would have looked only at the client ID and client secret and would have considered only the identity of the client application itself rather than the consent options of the resource owner implied by the authorization code. In other words, the *grant_type* parameter is used to disambiguate the flow the client expects the authorization server to perform.

The request could also include the *audience*. However, in this particular case, it is redundant. The authorization code has been granted in the context of that *audience*, and the authorization server knows it, hence there's no need to provide it again in this request. Adding the audience to the request can be beneficial for extra clarity: for example, this helps to interpret what this request is for while examining a network trace without the need to correlate it with the earlier messages that led to this point.

Finally, the message contains a **redirect_uri** parameter. In this phase, the authorization server doesn't really have any opportunity to perform redirects, given that the client is talking to the authorization server via a direct channel. Rather, the *redirect_uri* is used as a security measure to prevent redirection URI manipulation - the authorization server will verify that the redirect_uri presented here is identical to the one provided during the authorization code request leg of the flow, preventing an attacker from performing URI replacement (see https://tools.ietf.org/html/rfc6749#section-10.6).

7. **Receiving the Access Token in the Token Endpoint Response**
   Assuming that the request is accepted by the authorization server and processed without issues, the grant concludes with a response message carrying the artifact originally indicated by the *response_type* in step 2 - Authorization Request, in this case, an access token. Here's a breakdown of the response message content:

   - The requested access token.
   - An ID token, in response to the presence of *openid* in the list of requested scope values.
   - The token type, which is always *Bearer* for the time being - as discussed in the token validation section.
   - The *expires_in* parameter, expressing the time through which the access token should be considered valid. Although, at times, the access token itself might contain that information and happen to be in a format that can be inspected, access tokens should always be treated as opaque by clients. As such, *expires_in* needs to be provided as a parameter in the response so the client can use that information (for example, to decide how long an access token should be cached).

Important

Access tokens should always be assumed and treated as opaque by client applications because their content and format are a private matter between the authorization server and the resource server. The terms of the agreement between the authorization server and the resource server can change at any time: if the client app contains code that relies on the ability to parse the access token content, even minor changes will break that code - often without recourse.
Imagine a case in which access tokens, initially sent in the clear, start being encrypted so that only the intended resource recipient can decrypt. Any client will lose access to the token's content. Client code relying on the ability to access the token content will irremediably break. In summary, avoid logic in client applications that inspects the content of access tokens. Examining a token's content in a network trace is perfectly fine for troubleshooting purposes, as the information will be consumed via debugging tools without generating code that can break in the future.

8. **Using the Access Token to Call the API**
   Once the client obtains the requested access token, it can finally invoke the API: all it needs to do is include the access token bits in a classic REST call. In this particular example, the call is a GET, but any REST invocation style is possible. The key feature in that message is the *Authorization* HTTP header, which exhibits the *Bearer* authentication scheme and carries the bits of the access token.

   The OAuth 2.0 Bearer Token Usage specification, the document describing how to use bearer tokens obtained through OAuth 2.0 for accessing resources, says that it's possible to place the token elsewhere in the outgoing request, for example, in the body of a call or even a request link, as a query parameter. Encountering clients that send tokens in the body is very rare. The use of the query string for sending access tokens is actively discouraged, as it has important security downsides. Consider the case in which your client is running in a browser: whenever a token is included in the query string, its bits will end up in the browser history. Any attack that can dump the browser history will also expose the token. Moreover, if the API call is immediately followed by a redirect, the query string will be available to the redirect destination host in the referral header: once again, that will expose the token outside of the normal client-resource exchanges.

For those and other reasons, it is reasonable to expect that the near totality of the API calls encountered in the wild that rely on OAuth 2.0 will use the Authorization HTTP header.

Authorization Code Grant and PKCE

The latest OAuth 2.0 Security Best Current Practice (BCP) documents suggest that every Authorization Code flow should leverage *Proof Key for Code Exchange* (RFC 7636), an extension to the authorization code grant meant to protect Authorization Code from being stolen in transit. PKCE was originally devised for public clients, where it performs essential security functions that we'll describe in detail in the next chapter. We have chosen to keep this section light and to defer introducing PKCE in the next chapter, as you will be more familiar with the original grants, and it will be easier to add PKCE as an incremental step. However, we wanted to point out the BCP guidance already here so that if you read about it elsewhere, you'll know what it is all about.

Sidebar: Essential Authorization Concepts and Terminology

OAuth 2.0 offers a delegated authorization framework. Unfortunately, developers often disregard the "delegated" part and attempt to use OAuth primitives and flows to solve pure authorization scenarios that the protocol hasn't been explicitly designed to address. The outcome is solutions that might appear to work in toy scenarios, but fall short as soon as the approach is applied in more realistic settings.

For that reason, it is a worthwhile investment to spend a few paragraphs discussing essential concepts and terminology in authorization, spelling out explicitly their relationship with OAuth - and in particular, what is part of OAuth and what is instead a property of the underlying resources we are exposing.

Permissions

Imagine that you want to expose programmatic access to an existing resource. Depending on the nature of the resource, varying sets of operations can be performed on or with it. In the context of a document editing system, users will be able to see, read, comment on, or modify documents. An API that facades a printer might expose the ability to print in black and white or in color. Any kind of resource will have a set of permissions that make sense for that particular resource and that can be allowed or denied for a particular caller. A **permission** is just that, a statement describing the type of things that can be done with a resource: *document:read, document:write, print:bw, print:color, mail:read, mail:send,* and so on.

Permissions describe intrinsic properties of resources, which exist regardless of how those resources are exposed. OAuth 2.0 solutions might surface them if they are useful in the context of a delegated authorization scenario involving those resources. Still, in the general case, permissions exist in their own right and will be used outside of OAuth as well.

Privileges

A **privilege** is an assigned permission: it declares that a particular principal (say, John) can perform a certain operation on a given resource (say, calling the printer API to print in full color).

As was the case for permissions, the concept of privilege exists independently of OAuth (or any other higher-level protocol, for that matter). For example, the framework necessary to describe privileges needs primitives for principals (users and apps to whom permissions might be assigned) that OAuth 2.0 does not define.

The existence of permissions and privileges applied to a set of resources will influence the behavior of OAuth 2.0 solutions based on those resources, but how that will happen is not described directly in the protocol and messages defined in the OAuth 2.0 specification.

Scopes

Finally, we get to talk about an OAuth primitive. In the case in which a resource needs to be exposed in the context of a delegated authorization solution, the **scope** is the primitive that enables a client application to request exercising a user's privilege for a particular permission for a given resource. The mechanism that the client uses for expressing this to the authorization server is by including the scopes corresponding to the permissions being requested in an authorization request. When used with this semantic - that is, lists of permissions for a given resource - scopes are used to define the subset of user privileges that a client application wants to exercise on behalf of the user. Note that the scopes can be used for other purposes: we have seen examples of that in the case of *openid* (requesting the presence of an extra artifact, in that case, the ID token) or *profile, email* (influencing returned content).

Effective Permissions

We are finally ready to piece together how all those concepts interact with each other.
Consider a classic delegated authorization flow in which a client requests the authorization server to access a resource. In particular, the client specifies what permissions will be required for the operations it intends to perform on the resource. Upon receiving the request and authenticating the user, the authorization server will typically prompt the user to grant the app delegated access to the corresponding permissions. The user granting consent through that prompt is effectively saying, "Yes, I'm okay with this particular client exercising on my behalf the privileges being requested".

Say, for example, that the client implements an email solution, and the permission it requests is *mail.read*. The scope requested is mail.read and the access token being returned will include (by value or by reference, depending on the format) *mail.read*.

Once the client obtains the access token, it will use it to call the API and request to read a list of email messages. Upon receiving and validating the access token, the middleware protecting the API will verify that the scope it carries includes *mail.read*, the permission required by the API to perform the read operation requested and allow the request to move along.

But the authorization checks aren't over yet! Imagine that the client requests the list of emails from the inbox of a user different from the user who granted consent and obtained the access token. Should the API allow the request to succeed? Of course not! Scopes do not create privileges where there are none. Scopes can grant a client a subset of the privileges a resource owner has on a resource but can never add privileges the resource owner didn't have. T**he effective permissions are the intersection of the privileges a resource owner has and the scopes that have been granted to the client.** The effective permissions represent what a client can actually do, and that can be a subset of what's declared in the scopes. You always need to check at runtime whether the scopes represent something the resource owner can actually do for the resource being accessed.

Also, note that there is no guarantee that the privileges the resource owner had at the moment of granting consent will be preserved forever. Hence, even if your authorization server conflates scopes and privileges (for example, by only allowing a user to consent if they possess the corresponding privileges), nothing prevents some of those privileges from being revoked at a later time. This makes it necessary for the API to check rather than just relying on the scopes in the incoming access token. This is one subtle point that is often misunderstood in the context of OAuth.

Note that OAuth can also be used for application-to-application flows, in which no user is involved. The client obtains an access token for a resource from the authorization server only through its own client credentials, as opposed to requesting access on behalf of a resource owner. You could say that in those scenarios, the client application itself is the resource owner: there is no delegation, so there's no need for scopes to limit the privileges involved. We will study the corresponding OAuth 2.0 grant, the *Client Credentials* grant, in a later section of this chapter. In this case, it's not completely clear how permissions are expressed, as the core OAuth 2.0 specifications don't provide any mechanism to express assigned privileges (though there is a new specification, the JWT Profile for OAuth 2.0 Access Tokens, that does introduce some guidance about that). Regardless of the implementation details of how those privileges are expressed, this is a case in which privileges are actually carried in the token. There might be other cases where the authorization server includes user privileges, roles, group memberships, and other authorization information in the access token. Those cases are all valid and represent real, important scenarios. However, they aren't described by the specifications we are studying in this book, so we will not add further details here.

Finally, consider that although scopes often map to permissions, that is not always the case. Remember the *openid* scope? Its presence in a request just causes an ID token to be included in the response from the authorization server. Or think about the *profile* scope, which, when added to a request, causes the ID token to include claims that wouldn't be present otherwise. So it's easy to map between permission and scope. Scopes do correspond to permissions in many common cases, which might erroneously create the belief that scopes and permissions are the same concepts, but in fact, it's important to remember that they aren't.

The Refresh Token Grant

Let's now go back to grants. I mentioned this in passing earlier: tokens typically have an expiration time. They have an expiration time because a token caches a number of facts and user attributes, and those facts might change after the token has been issued.

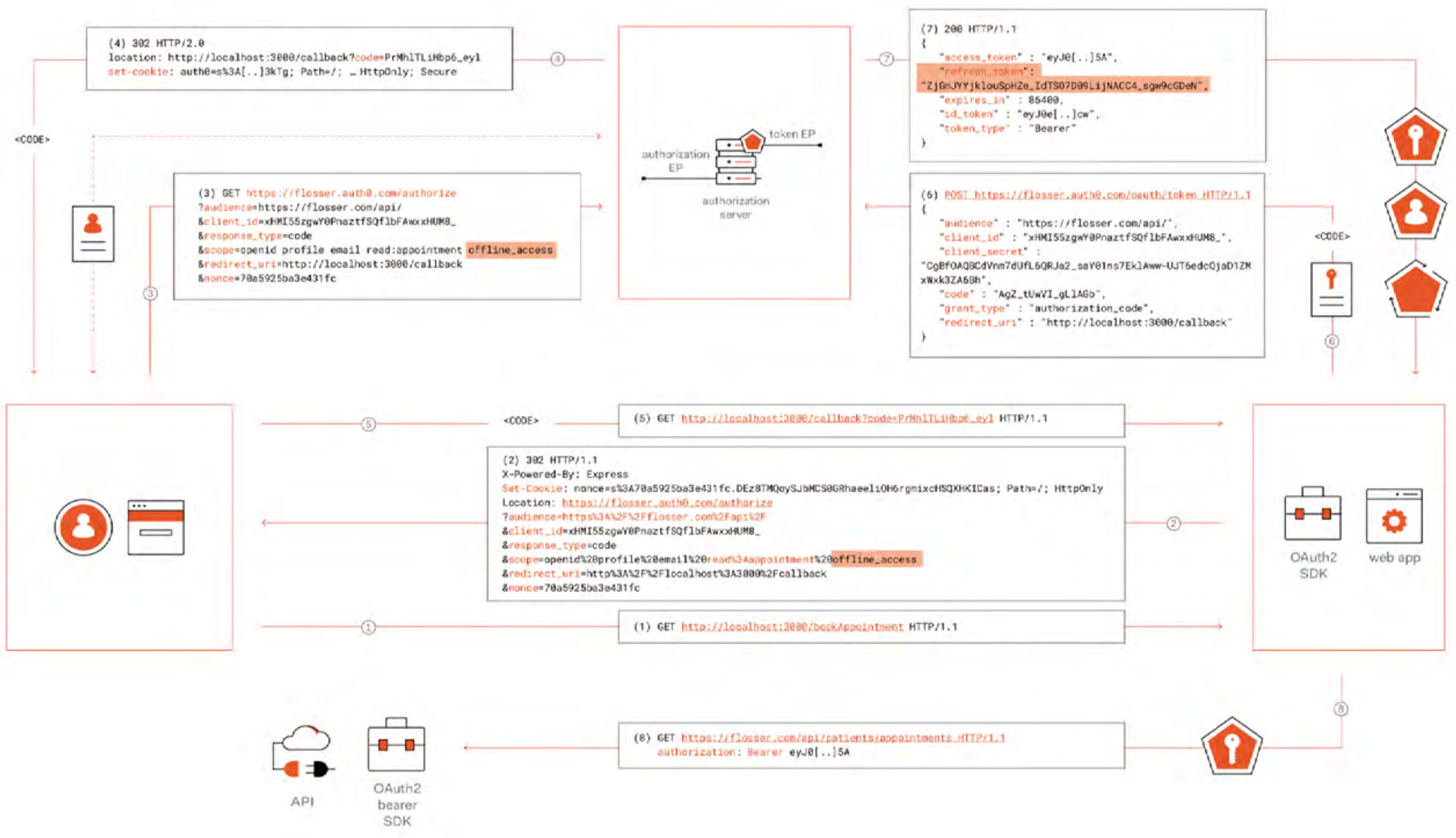
Also, the ability of a client to obtain a token at a given time doesn't guarantee that the same client will be able to get the same token in the future. For example, the resource owner might visit the authorization server and revoke consent for that client to obtain tokens with the scopes previously granted. This makes the content of any previously issued tokens obsolete as they no longer reflect the current situation.

The idea is that by endowing tokens with a short duration, we ensure that the client cannot really use them (and hence, the information they cache) for too long. Upon token expiration, clients will be forced to call back home and repeat a request to obtain a new token. This new request creates the opportunity for the authorization server to issue a new token containing up-to-date information or refuse to issue a new token if conditions have changed (e.g., the user account has been deleted from the system).
The shorter the token validity interval, the more up-to-date the issued information will be. Solutions typically seek compromises that balance the token's validity interval with performance and traffic considerations.

Of course, this brings another challenge: although we do want up-to-date information, we don't want to give users a bad experience to achieve that. The user should be blissfully unaware of all the low-level mechanisms unfolding behind the scenes to achieve those updates. We need to empower clients to renew tokens in a way that does not impact the user experience. OAuth solved this by introducing a new artifact, the **refresh token**, and associated grants, which are used to handle token renewals without displaying prompts.

The first step in working with refresh tokens is to request one. The OAuth 2.0 core specification doesn't define a mechanism to request refresh tokens, leaving the decision to issue one to individual authorization servers. However, OpenID Connect does define a mechanism to request refresh tokens, and the result is that a large number of OAuth 2.0 authorization servers adopt that mechanism as their main (or even only) way of requesting refresh tokens.

Let's revisit the authorization code grant examined in an earlier section and add a few small changes, as shown in Figure 4.3.



```
(1) GET http://localhost:3000/bookAppointment HTTP/1.1
```

```
(2) 302 HTTP/1.1
X-Powered-By: Express
Set-Cookie: nonce=s%3A70a5925ba3e431fc.DEz8TMQoySJbMCS0GRhaeeliOH6rgnixcHSQXHKICas; Path=/; HttpOnly
Location: https://flosser.auth0.com/authorize
?audience=https%3A%2F%2Fflosser.com%2Fapi%2F
&client_id=xHMI55zgwY0PnaztfSQflbFAwxxHUM8_
&response_type=code
&scope=openid%20profile%20email%20read%3Aappointment%20offline_access
&redirect_uri=http%3A%2F%2Flocalhost%3A3000%2Fcallback
&nonce=70a5925ba3e431fc
```

```
(3) GET https://flosser.auth0.com/authorize
?audience=https://flosser.com/api/
&client_id=xHMI55zgwY0PnaztfSQflbFAwxxHUM8_
&response_type=code
&scope=openid profile email read:appointment offline_access
&redirect_uri=http://localhost:3000/callback
&nonce=70a5925ba3e431fc
```

```
(4) 302 HTTP/2.0
location: http://localhost:3000/callback?code=PrMhlTLiHbp6_eyl
set-cookie: auth0=s%3A[..]3kTg; Path=/; … HttpOnly; Secure
```

```
(5) GET http://localhost:3000/callback?code=PrMhlTLiHbp6_eyl HTTP/1.1
```

```
(6) POST https://flosser.auth0.com/oauth/token HTTP/1.1
{
   "audience" : "https://flosser.com/api/",
   "client_id" : "xHMI55zgwY0PnaztfSQflbFAwxxHUM8_",
   "client_secret" :
"CgBfOAQBCdVnm7dUfL6QRJa2_saY01ns7EklAww-UJT6edcQjaD1ZM
xWxk3ZA6Bh",
   "code" : "AgZ_tUwVI_gL1AGb",
   "grant_type" : "authorization_code",
   "redirect_uri" : "http://localhost:3000/callback"
}
```

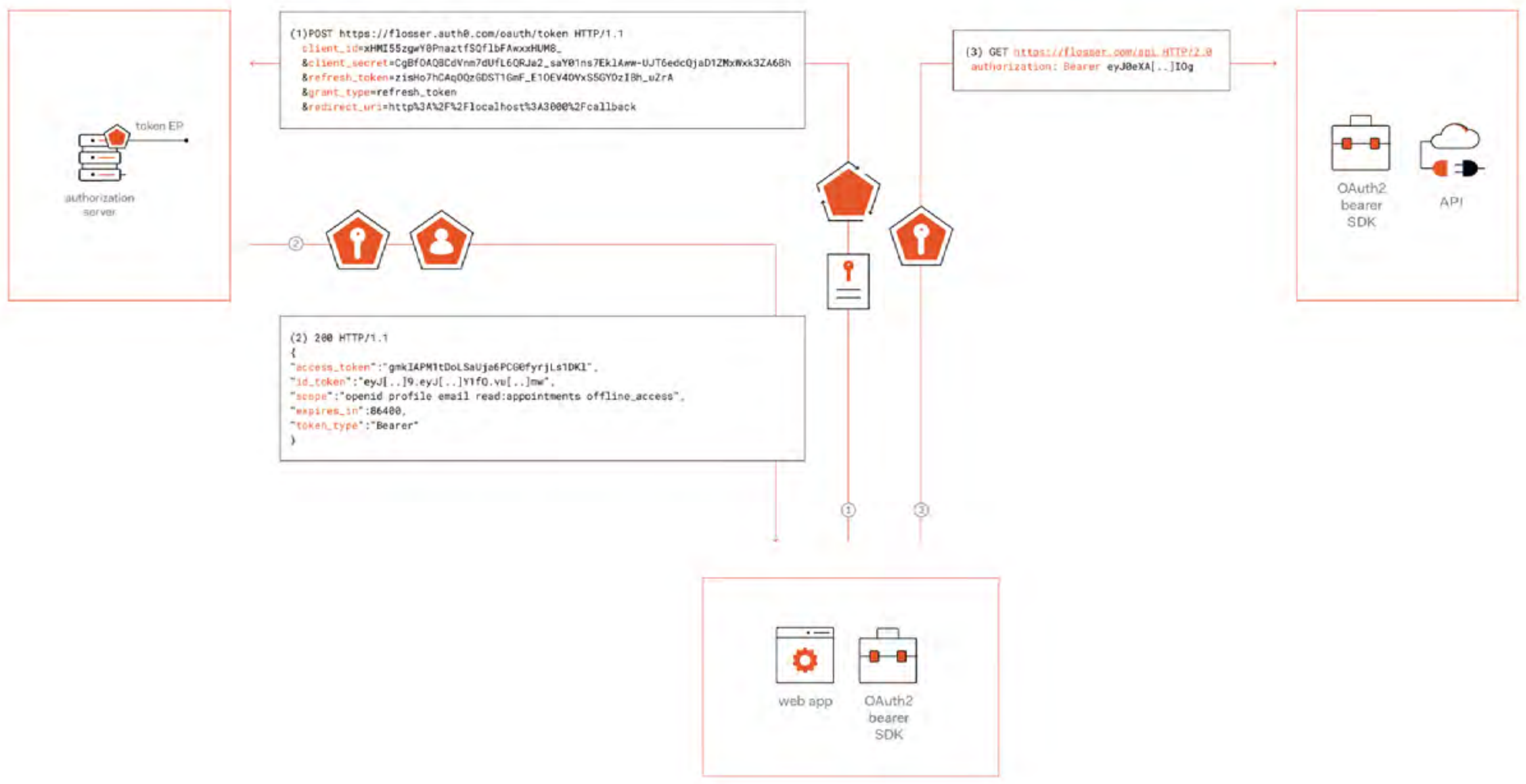
```
(7) 200 HTTP/1.1
{
   "access_token" : "eyJ0[..]5A",
   "refresh_token":
"ZjGnJYYjklouSpHZe_IdTS07D09LijNACC4_sgw9cGDeN",
   "expires_in" : 86400,
   "id_token" : "eyJ0e[..]cw",
   "token_type" : "Bearer"
}
```

```
(8) GET https://flosser.com/api/patients/appointments HTTP/1.1
    authorization: Bearer eyJ0[..]5A
```

Figure 4.3

The original message in step 3 carried the list of scope values the client required to request an ID token with rich attributes content (*openid*, *profile*, *email*) and the access level required for the operations the client intends to perform (*read:appointment*). The message in step 3 in Figure 4.3 contains an extra scope value, *offline_access*. This is a scope value defined in the OpenID Connect core specification: its presence in a request asks an authorization server to include a refresh token in its token endpoint response alongside all the other artifacts (in this case, an ID token and an access token). In particular, the validity of that refresh token will extend beyond the duration of the authentication session within which it has been issued. Don't worry if that's not very clear for now. We'll expand on what that means later in this section.
If you observe step 7 in the diagram, you'll see that, as expected, the authorization server returns a refresh token along with the usual access token and the ID token.

Now the client has a refresh token in its possession. Let's take a look at how the client uses it, and in particular how the refresh token makes it possible to get new access tokens without prompting the user again. The entire flow occurs on the server side, as it entails the client (in this case, a web app whose code runs on the server) connecting directly to the token endpoint of the authorization server. The browser, used to send the request and drive the interactive portions of the transaction, is now entirely out of the picture. Follow the numbered steps in Figure 4.4.



```
(1)POST https://flosser.auth0.com/oauth/token HTTP/1.1
  client_id=xHMI55zgwY0PnaztfSQflbFAwxxHUM8_
  &client_secret=CgBfOAQ8CdVnm7dUfL6QRJa2_saY01ns7EklAww-UJT6edcQjaD1ZMxWxk3ZA6Bh
  &refresh_token=zisHo7hCAqOQzGDST1GmF_E1OEV4OVxS5GYDzIBh_uZrA
  &grant_type=refresh_token
  &redirect_uri=http%3A%2F%2Flocalhost%3A3000%2Fcallback
```

```
(2) 200 HTTP/1.1
{
"access_token":"gmkIAPM1tDoLSaUja6PCG0fyrjLs1DKl",
"id_token":"eyJ[..]9.eyJ[..]Y1fQ.vu[..]mw",
"scope":"openid profile email read:appointments offline_access",
"expires_in":86400,
"token_type":"Bearer"
}
```

```
(3) GET https://flosser.com/api HTTP/2.0
 authorization: Bearer eyJ0eXA[..]IOg
```

Figure 4.4

1. **Refresh Token Redemption Request**
   The first leg of the grant takes the form of a typical token endpoint request analogous to the code redemption request described earlier in the chapter.
   Examining the request, you'll encounter the following parameters:
   - The usual *client_id*
   - The *client_secret*. This is a confidential client, so requests to the token endpoint require the client app to identify itself.
   - The new *refresh_token* parameter, which carries the refresh token bits received earlier.
   - The grant_type. As mentioned earlier, every request to the token endpoint must specify the grant the client intends to use. In this case, the parameter value is *refresh_token*.
   - The *redirect_uri* parameter, included for the same security reasons specified in the code redemption flow description.

2. **Refresh Token Response**
   The authorization server response returns a new access token, a new ID token (because the original request included *openid*), and the list of scopes that were granted when the refresh token was obtained, in this case, during the *Authorization Code* grant.
   The authorization server returns the list of granted scopes because the client might not really know what this particular refresh token was originally granted with or if the conditions at the authorization server have changed since its original issuance. Furthermore, the client can request a certain list of scopes, but the authorization server can always decide to return a subset of those scopes. In that case, if the authorization server wouldn't return the list of scopes that have been granted in the context of this particular refresh token redemption, the client would have no way of knowing. Even if it remembered the ones originally requested, there would be no guarantee that such a list would be accurate. Remember that the client is bound to consider the access token as opaque, so it cannot simply look into the access token to find out.

   In this particular case, the authorization server does not return a new refresh token alongside the access and ID tokens.
   The client is expected to hold on to the refresh token bits it received on the first flow and keep using it until expiration.

There are various scenarios in which the authorization server does include a new refresh token at every refresh token grant. The most notable case is in the context of a security measure called *token rotation*.

Token rotation guarantees that whenever you use a refresh token, the bits of that particular refresh token will no longer work for any future redemption attempts. Every use of a refresh token will cause the authorization server to invalidate it and issue a new one, which will be returned alongside the refreshed access token. Clients need to be ready to discard old refresh tokens and expect to store new ones at every renewal operation.

Any attempt to use an old refresh token will cause the authorization server to conclude that the request originator stole it. That might trigger protective measures, such as invalidating all the other tokens created in the same authenticated session in case the leak indicates a compromised application. Note that this measure might be overkill for confidential clients, where use from legitimate clients is enforced by requiring applications to use their *client_secret* when redeeming refresh tokens. However, it is extremely useful for public clients, where apps can redeem refresh tokens without exhibiting any app credentials. More details about this will be discussed in the next chapter on native and mobile clients.

3. **Calling the API**
   The new access token will be used exactly in the same way as the old one: all the considerations about calling API according to the OAuth 2.0 Bearer Token Usage specification apply.

Some Considerations on Refresh Tokens

The fact that a client requests a refresh token by including the scope *offline_access* signals to the authorization server that the resulting refresh token's lifetime will be decoupled from the lifetime of the authenticated user session within which the grant was performed. In other words, whether or not a user is signed in to an application via the front channel doesn't really matter with respect to whether the same application can redeem a refresh token.

Also, the fact that the app can still use a valid refresh token doesn't say anything about whether there's an active sign-in session for the user that helped obtain that refresh token in the first place. The two things are completely separated.

The scenario that *offline_access* is meant to support is the one I described at the beginning of the chapter, where a user wants to schedule a tweet to be published at a future time regardless of whether the user will be signed in at that time or otherwise. In more general terms, it addresses the case in which an application might need to obtain a valid access token to invoke an API even if no user is present to tend to interaction requests. One common mistake developers make is to interpret the ability of an application backend to redeem a refresh token as proof that the user still has a session. Per the above explanation, this is a dangerous mistake that can lead to resurrecting already expired or terminated sessions via sign-out, making front-channel session management ineffective.

When developing applications that need to invoke APIs even without an active user session, the app clearly needs to persist refresh tokens so that they are available independently of the presence of an interactive session. Even for cases in which API calls are scoped to the interactive session lifetime, tokens need to be saved somewhere other than in memory if you want to spare users from going through token acquisition flows in case the webserver memory recycles. Of course, persisting refresh tokens (and tokens in general) requires caution. It's important to make sure that tokens are stored per user to prevent the possibility of a user ending up accessing and using the refresh tokens associated with another user. That's just the same basic hygiene required to enforce session separation, but when it comes to tokens, following best practices is all the more critical given the high impact of identity mix-up and the complications that derive from persisting user data beyond the interactive session lifetime.

To close the topic of refresh tokens for this chapter, here's a last recommendation. Even if you know the expiration time associated with a refresh token, you should still not rely on that in your code. There are many reasons for which a refresh token might stop working, regardless of its projected expiration. For example, a user could revoke consent, immediately invalidating refresh tokens issued on the basis of previous consent. Another example: a resource server might change policy and establish that, from that moment on, it will only accept access tokens obtained via multi-factor authentication. This renders any refresh token obtained with a single-factor session unable to get viable access tokens and forces the client to reobtain a new refresh token via multi-factor authentication. Again, all this may happen regardless of the declared expiration of the original refresh token. For all those reasons, it is prudent to develop client code assuming that a refresh token might stop working at any time, and embed appropriate error management and remediation logic upfront.

Sidebar: Access Tokens vs. ID Tokens

You now had the opportunity to see both access tokens and ID tokens in action. Just as important, you learned about the reasons for which both artifacts have been introduced by OAuth 2.0 and OpenID Connect in the first place. It is worth stepping back for a moment and summarizing the differences between the two token types, as confusion about when to use what is one of the most common challenges you'll encounter as an identity practitioner.

Access Tokens Recap

Access tokens are artifacts meant to enable a client application to access a resource, typically on behalf of a resource owner, bestowing the client application with delegated authorization. As discussed, there is no token format mandated by OAuth 2.0.

Earlier, we discussed the implications of the common topology where the authorization server and resource server are co-located. This topology allows them to access shared memory and makes using a format for access tokens unnecessary.

Conversely, consider an authorization server separated from the resource servers, as with identity as a service offering like Auth0, where the same authorization server is shared by multiple resource servers owned by different companies. This scenario can really benefit from agreeing on a format and using it to validate incoming tokens, even if the protocol doesn't offer anything out of the box. The use of JWT as a format for access tokens is so common that it led me to drive a standardization effort to define an interoperable profile for it.

At the cost of being pedantic, it should be stressed that, as a client app developer, you should never write code that inspects the access token content. The fact that, in some cases, you might know that a specific token format is being used doesn't change this. The reasons why it's not a good idea are more about the contracts between the client, resource, and authorization server. In fact, it will often happen that you have a chance to look inside an access token, and the situation might change at any time. The format used in an access token is a matter agreed upon by the resource server and the authorization server, and the details can change at any time at their discretion without informing the client. Any code predicated on assumptions about the access token content will break as soon as those assumptions no longer hold, and on occasions without any remediation. Think of information being removed or the content being encrypted so that no entity but the intended recipient of the access token can inspect it. Although it is legitimate for a developer to read whatever information is available during troubleshooting, including the content of captured tokens, developing code that does so routinely will very often result in downtimes and serious production problems.

ID Tokens Recap

ID tokens are designed to support sign-in operations and, optionally, make authentication information available to clients. They don't contain any delegated authorization information (though nothing prevents implementers from extending the default claims set described in the specifications with their own custom values). ID tokens come into play during user sign-in, and clients can use them to learn about what happened during the authentication flow. Whereas clients should really not inspect access tokens, as discussed in detail just a few paragraphs earlier, clients must look inside ID tokens - that's part of the validation step described in the Web Sign-In chapter and mandated by the OpenID Connect core specification.

One of the most common points of confusion about ID tokens is whether they can be used for calling APIs. The short answer is that they shouldn't. Let's invest a few moments to understand why people attempt that and why it's generally not a good idea.

ID tokens are designed to support sign-in operations. The client app is simultaneously the requestor and the recipient of the ID token: once the client has received the token, it has reached its intended destination and isn't meant to travel any farther. All the client needs to do with it is validate it and extract user attributes when they are present. Both are operations that can be done locally, thanks to the fact that ID tokens have a fixed format, and the OpenID Connect specification details how to perform validation. The ultimate proof that the ID token shouldn't leave the client app lies in the *aud* claim, formalizing that the client app is the intended recipient by carrying its *client_id* value. We have discussed all this in Chapter 3, Anatomy of an ID Token.

Nonetheless, there are real-world situations in which client apps use ID tokens to invoke APIs. Often, that is due to designers not fully understanding the underlying protocols, and in particular, the role of the *audience* claim. For them, a JWT is a JWT, and an ID token is often easier to obtain as it doesn't require registering APIs, defining scopes, and adapting validation techniques to each specific authorization server requirements. For example, some will not use JWT as the format for access tokens and will require supporting introspection calls. Some others might not be designed to protect third-party APIs at all; hence, API registration and access token issuance and validation features are not offered, but ID tokens are still issued for sign-in purposes.

In general, using ID tokens to invoke API has issues. The main problem goes to the heart of why we have audiences in the first place. An API receiving an ID token can only verify that the token was issued for that particular client: there's nothing in the token saying that it was issued with the intent to call this particular API. Besides the practical issue of being unable to insert ad-hoc claims for that particular API, there are serious security concerns: a leaked ID token can now be used not just to access the client, but also to invoke this API and all the other APIs following the same strategy.

Whereas properly scoped tokens would contain the blast radius of a leak event (an access token scoped to API A can only be used with A), many APIs accepting an ID token means that they would all be compromised at once. This also makes it really hard to maintain separation between APIs: if both A and B accept ID tokens, that means that when the client calls A, A can turn around and use the same token it received from the client to invoke B. Although that might be acceptable at times, in the general case, this should never happen as a side effect.

Lastly, I will mention that the use of ID tokens for calling APIs cannot be secured by sender constraint, as the protocols supporting it won't provide any mechanism to associate the ID token to a channel between the client and the API.

For the sake of exhaustiveness, I want to acknowledge a particular situation where using ID tokens to call an API might not be disastrous, though it's never as good as using access tokens. Consider the case in which the client app and the API in itself happen to be the same logical application. That's the scenario commonly described as a "firs- party app", where both ends have the same owner and are tightly coupled to implement a given solution. Think of a social network API and its client app, for example. In this case, the solution won't strictly require delegation, the incoming token will likely be expected to identify the user, and the tokens issued to that client won't be accepted by any API other than the first-party one (if you exclude cases where individual app owners decide to accept them anyway, which are outside the control of the first-party solution developer anyway).

From the end-user perspective, the client+API ensemble constituting the solution is a logical whole - my experience of using my Twitter account through the Twitter app doesn't usually require any special consent where the APIs are explicitly called out. In that case, one could argue that the component of the app requesting the token and the component implementing APIs are, in fact, the same entity, which could be represented by the same identifier - hence, here's the crucial step, targeted by a token with the same audience… just like an ID token.

Once, in front of a beer, one of the authors of the OpenID Connect specification told me that an ID token is just an access token with specialized semantics. That said, it's still generally not worth it to ever use ID tokens for calling APIs. Although narrowly defined first-party scenarios do exist, those would still be better off when implemented with access tokens (think about sender constraint limitations mentioned above) and the risk of overreaching and using the ID token in ways that expose you to serious security risks is just too great. I mentioned this particular case here because you are likely to encounter that approach in the wild if you work in this space long enough, and I wanted to empower you to understand the nuances and point of view of the people following that approach: however, the best practice remains using access tokens for calling APIs. If you need JWT access tokens, use the aforementioned JWT profile for OAuth 2.0 access tokens.

ID Tokens and the Back Channel

OpenID Connect offers multiple different ways of signing in. The one we studied in the preceding chapter leverages the front channel. It relies on the Implicit flow (that is, issuing an ID Token directly from the authorization endpoint) plus form post (transmitting the token to backend-hosted logic, as it is the norm for redirect-based apps). That flow just happens to have the least number of moving parts, as it doesn’t require the client app to obtain, manage, and use a client secret. The flow has more or less the same security characteristics as traditional protocols such as SAML or WS-Federation, which are still widely used in mission-critical, high-value scenarios.

The *Authorization Code* grant we just studied in this chapter for calling the API can and is commonly used for performing sign-in operations - by obtaining ID tokens following the same steps we studied for requesting an access token. Say you are in a scenario in which, for some reason, you don't want to disclose the bits of the ID token to the user’s browser. Using the *Authorization Code* grant, you can make everything take place on the server side. You can just perform an *Authorization Code* grant in the same way we did to get a token to call the API: you just ask for an ID token as well. Note, that’s exactly what we did in our API calling scenario by including the *openid* scope in the initial request. All we need to do to make that operation count as sign-in is validate that ID token and create a front channel session based on its content.

The notable difference from the front channel is that, given that the client obtains the ID token from a direct HTTPS connection with the token endpoint, there is no uncertainty about the source from which the ID token bits came from. The client knows for certain that the ID token comes directly from the authorization server, with no intermediaries that could have tampered with the content in transit. With origin and integrity verified, there is no need to validate the ID token’s signature. Think about it: if you were to validate the signature, you'd use the key you retrieved from the discovery document. And why do you trust that it is the right key? Because you retrieved the discovery endpoint over an HTTPS direct channel! The same assumptions hold for the ID token retrieval from a direct connection with the token endpoint, which is why the client can skip the signature verification.

What's very, very important to understand is that not having to verify the signature does NOT mean that the client is allowed to skip token validation! The client is still meant to validate audience, issuer, expiration times, and all the other checks that the OpenID Connect specification describes for the ID Token validation. The signature is only one of the many checks a recipient should perform to validate incoming tokens, even in the front channel case.

However, keep in mind that while having a direct HTTPS connection with the token endpoint assures you of the token's origin, it does not ensure that the token you receive is the one you requested. An authorization code injection may have occurred between the initial request and the exchange of the authorization code with the authorization server, and your application has no way of realizing it. This is why OAuth 2.1 recommends using PKCE with confidential clients, too.

Obtaining an ID token via the *Authorization Code* grant is technically more secure than receiving it through the front channel. However, this technique is more onerous, as it requires the client to obtain, protect and use an application credential - that has a management cost, associated risks (like forgetting a secret in source control), performance, and availability challenges (extra server calls). If your application only needs to sign-in users and doesn't have particular constraints about having tokens transit through the browser, the front channel technique works fine - as demonstrated by many years of successful SAML deployments using similar techniques to protect high-value scenarios. If you are indeed in a situation that calls for higher security or already performing API calls requiring the authorization code flow anyway, you might consider implementing sign-in via backchannel as described in this section.

The UserInfo Endpoint

A client requesting an ID token without specifying the *profile* and *email* scope values will receive a skeleton token stating that user X (as expressed by an opaque identifier, usually) successfully authenticated with issuer Y. The token also specifies the time and perhaps the authentication modes, and no other info - in particular, no user attributes.

There might be multiple reasons for which a client might opt for such barebone ID token content. For example, a client might want such a token to use an easy to set up front channel sign-in flow while avoiding disclosure of personally identifiable information (PII) to the browser. Alternatively, clients might go that route simply to reduce the size of transferred data on a network that doesn't have a lot of bandwidth or on a metered connection where bigger ID tokens might result in the user getting charged more for data use.

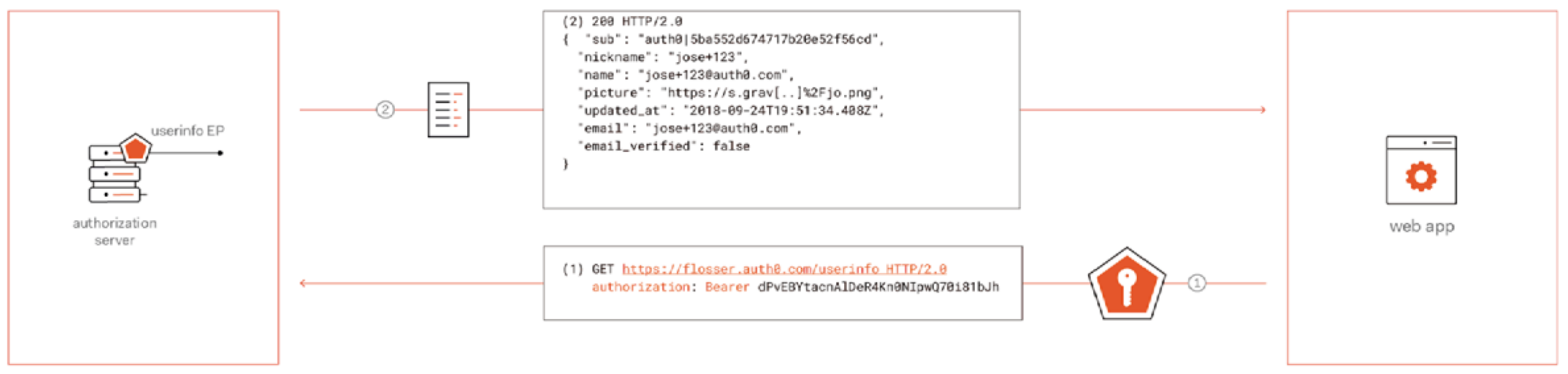
The good news is that clients can opt to work with barebone ID tokens and still gain access to user attributes when necessary. OpenID Connect introduced a new API endpoint, called **UserInfo** endpoint, which can be used for retrieving information about the user by presenting an appropriate access token - following the same OAuth 2.0 bearer token API calling technique studied earlier in this chapter. Whenever the client needs to know something about the user, whether it didn’t save the initial ID token or received a barebone one, it reaches out to the UserInfo endpoint using a previously obtained access token. It will receive what substantially is the content that the client would have gotten in an ID token requested with *profile* and *email* scopes.

The first chapter described the evolution that led from OAuth 2.0 to OpenID Connect. A key passage was about a particular way of abusing OAuth for simulating sign-in, where the ability to successfully call an API with an access token was considered proof enough for the client to consider a user signed in. That had several problems: access tokens could not be tied to a user in particular (very important if you are trying to authenticate, that is, to sign-in), could not be proven to have been issued as part of a sign-in operation for that app in particular, and could not be standardized given that every provider protected API of different shape (Facebook Graph, Twitter API, etc.).

The UserInfo endpoint resolves the first and third problems. The UserInfo response does provide information about the user who obtained the access token used to secure the call to begin with - and since it’s standard, generic SDKs can be built to work against it. That makes it possible for a client to implement pure OAuth 2.0 to retrieve user information in a standardized fashion.

It is very important to realize that, however, successfully calling the UserInfo endpoint is NOT equivalent to validating ID tokens and alone CANNOT be used to implement sign-in, it does NOT count as sign-in verification. Calling the UserInfo endpoint **only proves that the corresponding access token is valid and associated with the user identity whose attributes are returned**: it does NOT prove that the access token was issued for that particular client. OpenID Connect sign-in operations ALWAYS require validating an ID token, although, as we have seen in some circumstances, the signature check can be skipped from the validation checklist.
Another thing to keep in account when considering using the UserInfo endpoint from a confidential client is that all the discussions about the burden of using a secret apply here, as that's part of obtaining an access token.

After all that preamble, let's take a look at how an actual call to the UserInfo endpoint takes place. As usual, we are going to explain each step - please refer to the numbered messages in the diagram in Figure 4.5.



```
(2) 200 HTTP/2.0
{  "sub": "auth0|5ba552d674717b20e52f56cd",
  "nickname": "jose+123",
  "name": "jose+123@auth0.com",
  "picture": "https://s.grav[..]%2Fjo.png",
  "updated_at": "2018-09-24T19:51:34.408Z",
  "email": "jose+123@auth0.com",
  "email_verified": false
}
```

```
(1) GET https://flosser.auth0.com/userinfo HTTP/2.0
    authorization: Bearer dPvEBYtacnAlDeR4Kn0NIpwQ70i81bJh
```

Figure 4.5

1. **UserInfo Request**
   The scenario in the diagram assumes that the client has already obtained a suitable access token to call the UserInfo endpoint. Invoking the UserInfo endpoint is simply an HTTP GET request, attaching said access token in an authorization header.

   You might notice that in this particular network trace, the access token value looks different from all the other tokens shown in the diagrams so far. Whereas token values in earlier diagrams were always clipped for presentation purposes, and their shape suggested the classic JWT encoding, the bits on display here are the entirety of the access token and don't appear to follow any known pattern. That's because calling the UserInfo endpoint is precisely a scenario in which opaque, formatless tokens make sense. The UserInfo endpoint is co-located with the authorization server; there is no need for cross-boundaries communication. The entity that issued the access token in the first place is the same entity responsible for validating it during the UserInfo API call. That means that the two tasks can access the exact same memory space. In concrete terms, this means that the access token intended to access the UserInfo API doesn't need to be encoded in any particular format. It can literally be the identifier of a row in a database created at issuance time and can now be looked up at API invocation time or any other technique relying on shared memory.

   We cannot afford this luxury when the API being invoked is managed by a third party and hosted elsewhere. In this scenario, the parties involved are forced to rely on token validation based on formats, introspection, and, in general, techniques meant to accommodate the lack of shared memory between the entity issuing the token and the entity consuming it.

2. **UserInfo Response**
   The response returned by the UserInfo endpoint contains pretty much the same list of claims carried by an ID token obtained via a request that includes the *profile* scope.

The Hybrid Grant

The **Hybrid grant** is, as the name suggests, a mix of multiple flows into one. It combines a sign-in operation (getting an ID token from the front channel) and obtaining an access token for invoking an API from the client backend (by requesting and redeeming an authorization code). That saves network round trips, consolidates prompts and consent requests, and is, in general, a very efficient way of performing a sign-in operation while getting ready to invoke API at the same time. No diagram is shown for the hybrid grant, as you can easily piece it together yourself by combining the web sign-in flow diagram in the preceding chapter and the Authorization Code flow shown here. OpenID Connect is unique in this ability to mix and match sign-in and calling APIs and having entities playing both roles: a “resource”, as in something being accessed as part of the sign-in access, and a client, consuming other resources such as API. The fact that the app in OpenID Connect is always called a client, emphasizing the latter role and omitting the former, is a nod to its
OAuth 2.0 origins (and to the fact that “resource” in OAuth 2.0 is reserved for APIs).

The *Hybrid* grant is a really powerful tool that is commonly used in applications. In fact, today, it's pretty rare that an app will forever either only require sign-in or only call APIs. It's usually a continuum, and the availability of this grant makes it easy to add one functionality or the other by simply modifying either the Implicit plus Form Post grant or the Authorization Code grant.

Client Credentials Grant

In the last section of the chapter dedicated to invoking API, we will study the **Client Credentials** grant, a flow defined by OAuth 2 for cases where a client needs to get access tokens using its own programmatic identity, rather than doing so on behalf of a user. Unlike the grants we examined so far, the *Client Credentials* grant has no public client variant - it can only be performed by a confidential client.

All the flows examined so far for API are designed to grant clients delegated access to resources, that is to say, to enable clients to “borrow” some of the user’s privileges when accessing resources.

There are a number of situations in which clients need to operate as themselves rather than on behalf of a user. These are scenarios in which the application has an identity and direct resource privileges in itself. That class of scenarios doesn't require a user to be signed in or otherwise present. Even if a user happens to be signed in at that time of access, their privileges might not be the ones the client needs to exercise. A classic example of that scenario occurs when an application needs to perform an operation for which the currently signed-in user has no privilege. Imagine, for example, a Continuous Integration (CI) web app in which the final step of a build process is taking the binaries of a compiled product and saving them in a particular share that no user has access to.

One way to work around the problem would be to open the floodgates and give every user permission to access that share. That would preserve the CI's ability to call the share in delegated access mode. However, the risk for abuse would be very high: users might choose to exercise their privileges on that file share even outside of the CI process.
An alternative would be to give privileges for file share access *to the application itself*. In turn, the application can feature logic that determines which users should be able to write to the share. So, it can use its own write privileges to perform writing operations only for the appropriate user sessions and only within the limits of what the CI logic requires. Said in another way, by granting the application itself the necessary privileges to access a resource, the responsibility of determining who can do what is transferred from the authorization server to the application itself, which becomes the gatekeeper for the resource.

One common way of referring to the aforementioned pattern is to say that the application and the downstream APIs it accesses are defined as a **trusted subsystem**.

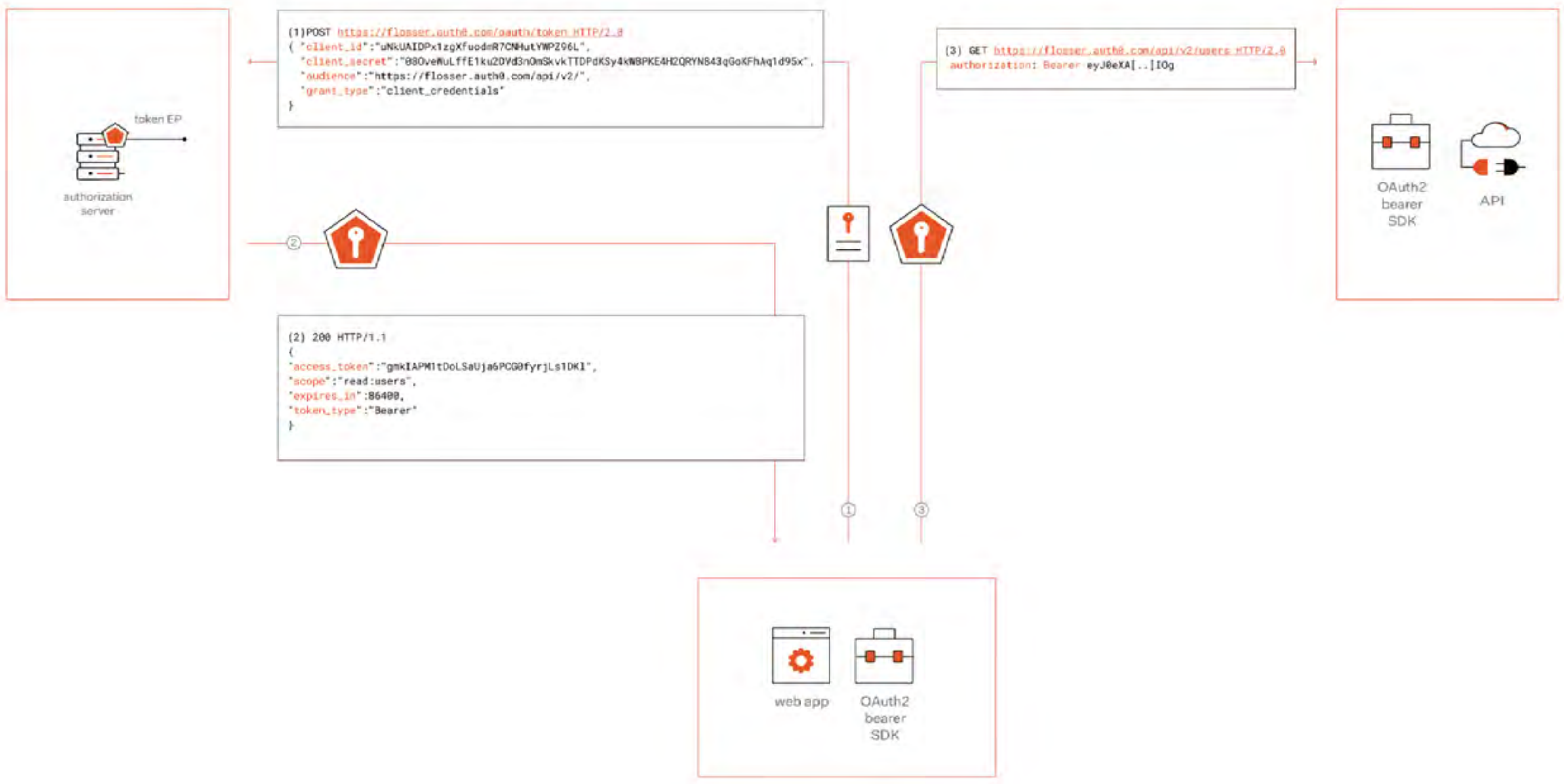
To use a real-world analogy, consider how a classic amusement park handles visitors' access. At the entrance, a visitor pays for a ticket and is given a bracelet or equivalent visible sign that the individual paid for access. This sign does not need to bear any indication of the wearer's identity. Once the guest is in, they can enjoy every ride without any further access control check other than the bracelet, broadcasting their right to be on the premises.

Similarly, once a user signs in with the CI web app, all subsequent calls to the downstream API will be performed as the web app itself, just in virtue of the fact that the user successfully signed in. In a way, you can think of this as a resurgence of the concept of perimeter. However, the big difference with traditional network perimeter is that the boundaries here are mostly logical (API's willingness to accept tokens issued to the CI app client) rather than physical (actual network boundaries).

This class of patterns is pretty common in the context of microservices, where a gateway validates the caller's identity. Once that check has been successfully performed, all subsequent calls from the gateway can be performed carrying tokens identifying the calling app rather than the user. The user information might still be required, but it doesn't strictly need to travel in an issued token.

As is the case with every confidential client flow, the critical point here is in putting particular care into provisioning client credentials and maintaining them, for example, by ensuring that no entity other than the application has access to its credentials. Another critical aspect of the scenario, not explicitly covered by the standards but of vital importance, is to carefully choose the privileges assigned to the application and the application logic exercising them. The least privilege principle remains a key best practice in this scenario.

Let's take a look at how the *client credentials* grant actually works on the wire. Please refer to Figure 4.6.



```
(1)POST https://flosser.auth0.com/oauth/token HTTP/2.0
{ "client_id":"uNkUAIDPx1zgXfuodmR7CNHutYWPZ96L",
  "client_secret":"080veMuLffE1ku2DVd3nOmSkvkTTDPdKSy4kWBPKE4H2QRYN843qGoKFhAq1d95x",
  "audience":"https://flosser.auth0.com/api/v2/",
  "grant_type":"client_credentials"
}
```

```
(2) 200 HTTP/1.1
{
"access_token":"gmkIAPM1tDoLSaUja6PCG0fyrjLs1DKl",
"scope":"read:users",
"expires_in":86400,
"token_type":"Bearer"
}
```

```
(3) GET https://flosser.auth0.com/api/v2/users HTTP/2.0
authorization: Bearer eyJ0eXA[..]IOg
```

Figure 4.6

1. **Access Token Request**
   The client application requests a token by contacting the token endpoint directly, similarly to what we have observed in the server-side segments of all the grants we have studied so far.

   In the sample scenario we have been discussing so far, the call is performed during a user session - however, that is entirely arbitrary. Remember that the *Client Credentials* grant only relies on the client's own identity rather than requesting delegated authorization from a user. So, from the OAuth 2.0 standpoint, the flow described here might just as well occur in a command-line tool, a long-running process, or, in general, any kind of application executed in a context where distribution and protection of client credentials are possible.

   The request is a customary HTTP POST, carrying the well-known *client_id*, *client_secret*, and *grant_type* (this time, set to *client_credentials*)

   Observing the body of the POST message, one notable difference from all the grants encountered so far is that the message for the token endpoint doesn't contain any artifact besides the *client_secret*. In contrast, the *Authorization Code* grant and the *Refresh Token* grant all included some other entity to redeem. Once again, this shows why the other flows are conceivable with public clients as well, whereas the *Client Credential* grant isn't possible without, well, client credentials.

   Here, it's opportune to stress that client credentials and the *Client Credentials* grant are two separate, distinct concepts. Client ID and client secret are the client credentials assigned to a confidential client application and are used to identify the client app in every grant whenever communication with the token endpoint occurs. The *Client Credential* grant is a grant that happens to require only the client credentials and no other artifact to be performed. It's easy to get confused when using the terms loosely: whenever you hear someone mentioning "client credentials", it's useful to be clear on whether they are talking about the grant or just about the client ID and client secret.

   One last observation on the request message: the *audience* parameter must indicate to the authorization server what resource the client requests access to. This information is necessary for authorization servers that can protect multiple source servers; hence, there's no default resource the authorization server can refer to. As mentioned in our earlier discussions about the *audience* parameter, the standard way of signaling that information to the authorization server is through the *resource* parameter defined in the resource indicators specification. At the time of writing, Auth0 doesn't support resource indicators.

2. **Token Response**
   The token endpoint response is entirely unsurprising. It carries back the requested access token, just as described for other grants. Of course, there is no *id_token*, given that the grant didn't entail user identity in any capacity.

   Notably absent is the refresh token, too. In this scenario, it would simply serve no purpose. The refresh token is meant to allow a client app to obtain a new access token to substitute an expired one without bugging the user with an extra prompt. However, there is no need to ask anything from a user here, as the client credentials are available to the client app at any time to request a new token.

Important note

The mechanism shouldn't be abused. Once a client requests and obtains an access token, it should keep it around (stored with all the safety measures the task requires) for the duration of its useful lifetime and use it whenever it needs to call an API. Discarding still-valid access tokens and requesting a new access token from the authorization server every time can be a costly anti-pattern at all levels:

- Security (every time credentials are sent on the wire, there's an opportunity for something to go wrong).
- Performance (network calls).
- Availability (possibility of being throttled, transient network failures).
- Money (various providers charge per issued token).

Note that, in this particular case, Auth0 uses *scope* to represent what the client can do. From what we said earlier about scopes, this is a bit controversial. Let's say that scopes normally restrain the set of privileges that the client can use from the user's privilege, and here, there is no user. Even if it does not appear quite appropriate, that's how Auth0 does it today. It just represents the privileges that have been granted to the client application. There is no real security risk because of this: if a resource owner would interpret the incoming scopes as the delegated authorization concepts we discussed so far, the power they'd confer to the caller would be *less*, not more. However, it's an exception that is important to be aware of.

3. **Calling the API**
   As expected, the call to the API occurs as usual, without any dependency on how the client obtained the access token being used to protect that call.

This completes our journey to understanding how to leverage OAuth 2.0 and OpenID Connect to invoke APIs from a traditional web app and, in general, any confidential client.

In the next chapter, we'll take a look at native clients: mobile clients and pretty much any application that an end-user can directly operate… and that isn't a browser.

Chapter 5

Desktop and Mobile Apps

It's finally time to touch on one of my favorite topics: how to secure applications meant to run on your desktop or mobile devices.

Public Clients

However, before I do that, I have to introduce yet another actor in our play: I want to spend some time describing what a public client is.

We have seen that a confidential client is defined as a client that has credentials and can use those credentials to prove its own identity to the authorization server regardless of the identity of a user. You guessed it: a public client cannot do that. Typically, it's because it's just hard to distribute credentials to, you know, public clients. And it's as hard as keeping them secret.

So, for example, imagine a situation in which you are installing an application from an application store on your mobile device. You are downloading the bits of this application, which will live on your device. There is no protocol as part of the application's distribution that also gives you a key representing that particular instance of this app.

But even if we could get such a key, it would become a secret specific to that app instance. If it's used to identify the client, like that client ID we used on the server, now you'd have an attack surface that basically leaves all the way to the pockets of a potential hacker.

As we said earlier, if you assign a credential to a website, I need to compromise the server to try to steal that credential. In contrast, here, the device is in my pockets. It's at my disposal and sometimes I can share it with others. I can install multiple applications without doing an accurate technical check. In other words, my device can be inadvertently exposed to malicious attacks more than a server. So, a key representing my particular instance of the app would be more than the client secret associated with a client ID, and in this scenario, that would make no sense.

One interesting part is that we might not care all that much about this limitation, mostly because when you're using such applications, the highest order bit is the user. So, if I'm using Slack on my phone and another colleague is using Slack on their phone, in the end, the authorization decisions are based on the fact that it's Slack. Sure, Slack might need a list of scopes, which have been granted. But the highest order bit is really the user and what the privileges of the user are.

The best scenario is to have some mechanism for preventing people from taking tokens and using them from a different device.In the absence of such a mechanism, we can take into account the fact that we don't have a secret and tune our authorization decisions accordingly.

One super important point here is that if a client ID occasionally looks obscure, i.e., it's too far to be human-readable, it does not mean it's a secret. It's not a secret at all. **A client ID is public**. You have to assume it's public. As a matter of fact, every identifier or credential distributed to such a client is public.

So, when you have a native application that is a public client, you have to assume that anyone can grab that client ID and pretend to be your application. That's by design; that's expected. So, you should never make authorization decisions on the server side based on the ID of a native client because that thing is just a hint. It's not really proof of anything at all. It's super important!

Native Applications and the Browser

Now how do we do this? We have seen that when we use the authorization server, OpenID Connect providers and similar, the typical way we use for interacting is through web pages of some kind which is super handy because we can change the UI at any time, and we can inject multiple authentication factors. We don't need it to really cache anything on the client. We don't need it to have a dedicated code on the client for doing prompts and similar.

But here, we have native clients with code living on a device. So, how can we interact with the authorization server? The trick is to open a little window on the browser whenever we need to do authentication. So, even if I'm a native client, I can always provide some kind of surface capable of rendering HTML, and I can use that surface to drive all interactions with the authorization endpoint. Once I'm done and receive the artifacts I want - tokens, calls, and similar - then I can take over from my code, close whatever I used as a browser, and just go ahead with my flow.

There are different ways of doing this. The traditional way apps did it at first, and now no longer recommended and unsupported by some authorization servers, is to use an embedded browser or an embedded WebView. An embedded WebView is a native component, such as a component of your operating system or your window management system, that you can place on the surface of your app, just like buttons, labels, and similar, and this thing will render HTML.

Doing this has risks. One particular risk is that an application can control everything that happens on its surface. So if it pops out a browser window that lives inside the application, and the user enters credentials, that application can record each and every keystroke, which is clearly dangerous. Say you are using an application that needs access to Facebook for user authentication. In my case, I'm a subscriber to the "New York Times", and I associated my subscription with my Facebook account. If Facebook's login page is embedded in a WebView, that app can intercept my credentials and impersonate me in other contexts.

The other problem is that this embedded WebView is by design isolated from whatever browser lives on the machine. As a result, you will get some inconvenience in the user experience.

Consider the app to read the “New York Times” mentioned earlier: whenever, for some reason, I'm not authenticated, I end up getting this little window saying, "Authenticate to Facebook".

When this happens in an embedded WebView, it doesn't matter that I have already signed in to the Facebook app; I have to sign in again. I get prompted for my username and password because the WebView is isolated from the device's cookie jar. That's extraordinarily annoying to the point that very often, I just close this thing and forget about it and remain ignorant because it's just a lot of work to have to enter this stuff.

Now, today’s mobile operating system providers supply the solution to this problem: a programmatic way of invoking the system browser from your applications. So, when a mobile app, such as an iOS or Android app, needs to get a token from your authorization server, you can use a system call that opens the system browser. The app switches the focus to a slice of the system browser: a Safari view controller on iOS or a custom Chrome tab on Android.

This is a view of the system browser with a single tab that has access to all the values, including cookies, and that, above all, is not in the application's memory space. It's the browser. At this point, the user can enter credentials, do MFA, and take advantage of existing cookie jars without leaking any of their credentials to the calling application. That's really powerful and super handy.

The thing is that it adds a bit of extra attack surface because like when you have your embedded browser, the communication between the browser and your application all happens in the memory space of your app.

So, say that you're doing the Authorization Code flow. I use the browser to get the authorization code, and then I pass the code to my application. I need to communicate the code from the browser to the app. That means that if someone is in the middle, say another app, they might intercept this code. Given that the app has no credentials - remember that this is a public client - whoever intercepts that code might use it and obtain tokens instead of me.

Meet the PKCE

To prevent someone from intercepting the authorization code while it's moving from the browser to my app, you can use a mechanism that substantially ties the request of the code to a secret created by the app on the fly.

The application must demonstrate knowledge of that secret at code redemption time. As a result, if anyone steals the code in transit, they will not be able to use it without knowledge of this secret. I'll show you in detail what that means.

So, when you use the system browser, you should not just use the Authorization Code flow but also add this mechanism to protect communication of the code. This mechanism is called Proof Key of Code Exchange (PKCE), which is pronounced "pixie", and is defined as an extension to the Authorization Code flow.

Desktop Applications and Browsers

Now, here is another controversial point. The best practices document on using OAuth in mobile apps substantially says what I just told you: You should use a system browser and protect communication between your application and the system browser using PKCE.

That document also tells you that you should do the same on desktop applications, i.e., applications running on your Windows, Mac, or Linux machine. Frankly, that's just not practical. That's to say, if you try to do the same for applications that run on the desktop, you might incur a few issues when you call the system browser. For example, you don't know what browser is installed on the machine. Also, this browser might not come up on top because you don't control the Z-order of the browser window, or the user can have multiple browser windows open, or they have only one window, but the application might run in a modal window.

Above all, if you really want to be compatible across multiple operating systems, in order to bounce the communication back to your application, you need to have a mini web server that runs locally on the machine. This web server listens to your redirect URI, receives the authorization code, and shoots it back to the app.

In other words, if your operating system does not have a mechanism comparable to what we have on iOS and Android to directly involve the system browser in the transaction, this makes the experience for the end-user really tough. It also complicates the flow and makes security people nervous because opening sockets on your machine is not fun.

In addition, when you are on a device with no browser whatsoever, you can't use this flow because it's all predicated on having a browser's availability - whether embedded or a system browser.

So, if you are developing a command-line application, you can only use this flow if you target a machine with a browser. Another grant - the *Device Authorization* grant - allows you to use a browser on a different device and close the cycle. But I won't go into the details here.

The Authorization Code Grant with PKCE

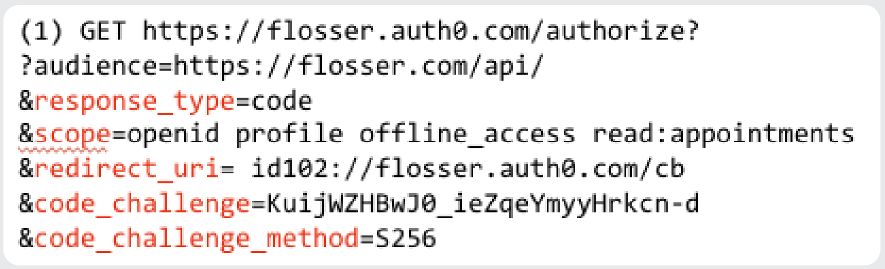
Let's look at how these things take place following the diagram shown in Figure 5.2. In this scenario, I have an API that I want to invoke. There is my usual authorization server with its good old authorization endpoint, token endpoint, and discovery endpoint.

On the client side, there is a lot of new stuff. We have our native application with the usual SDK for implementing OAuth and a cache for saving tokens. The system browser is a different app running within the same device. Now, let's go through the flow for our application to get a token for calling the API following the numbered steps.

1. **Authorization Request**

   In the case of native clients, I can't first hit the resource and then be redirected to the authorization endpoint because my app renders the UI. I don't rely on the server to redirect me to the app with the authorization request. So, I need to first do whatever steps are necessary to get my token and then call the service, which is why the diagram of this flow does not start with a line to the resource but with a line to the authorization server.

   The application uses the operating system API to invoke the system browser and make it talk to the authorization endpoint. Here is the request sent to the authorization server:



```
(1) GET https://flosser.auth0.com/authorize?
?audience=https://flosser.com/api/
&response_type=code
&scope=openid profile offline_access read:appointments
&redirect_uri= id102://flosser.auth0.com/cb
&code_challenge=KuijWZHBwJ0_ieZqeYmyyHrkcn-d
&code_challenge_method=S256
```

Figure 5.1

Actually, we are doing an Authorization Code grant, so you shouldn't be surprised to see the content of this request, at least for the most part.

The first parameter is the *audience*. We have seen what the *audience* represents, i.e., the particular resource we want to access. We have seen that it's specific to Auth0 and that an equivalent extension to OAuth 2.0 exists, although currently not supported.

We have the *response_type* parameter with the value *code*. We don't specify the response mode, so we know that we'll get it on the query string.

Then we added our list of scopes for the same API that we were calling earlier: *openid, profile, read:appointments*. We also ask for *offline_access*.

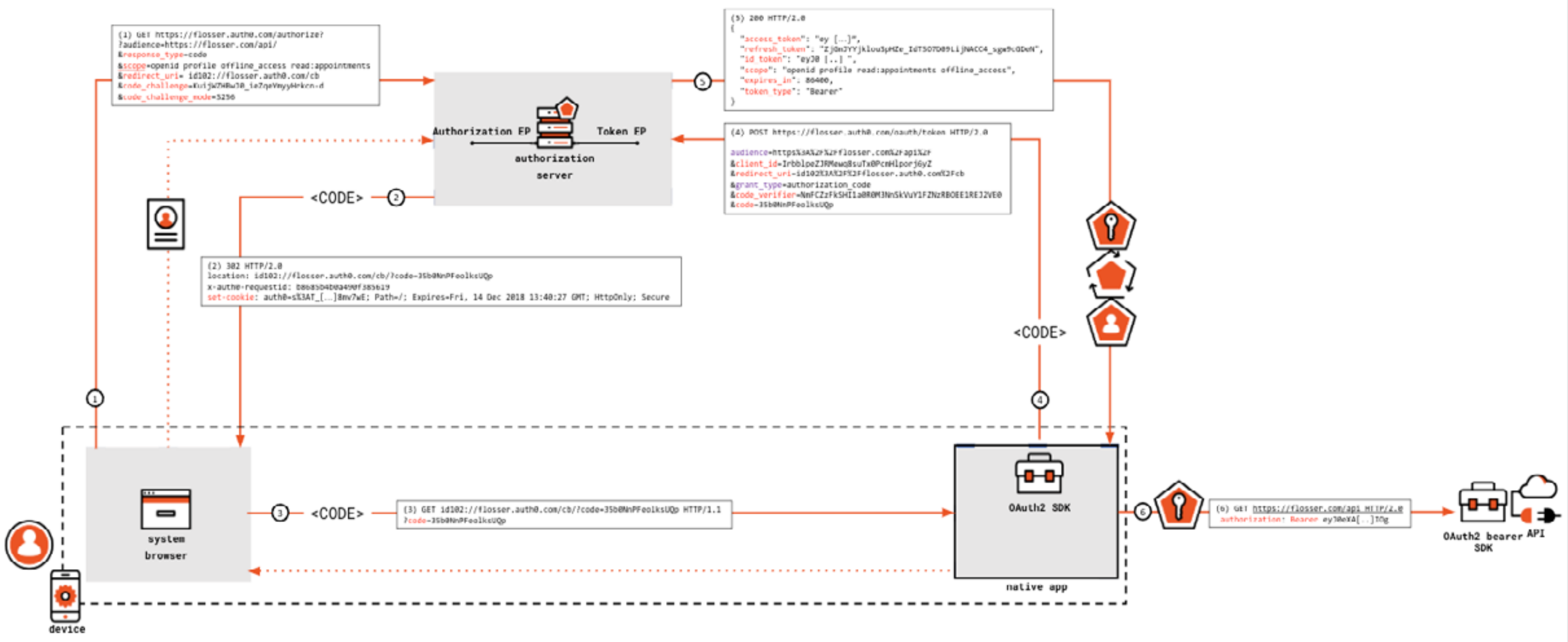
In the case of native clients, the refresh token represents, in some ways, your session because it's the main artifact you have under control and grants you the ability to get new tokens. So, when you sign out of a native client app, you also dispose of the refresh token.

The *redirect_uri* brings the first new thing: instead of having HTTPS, it has *id102*. This *id102* string is just a protocol handle that we invented when we provisioned this application on the operating system. This protocol handle represents our app. It tells the operating system that whenever it sees someone trying to follow a link that starts with *id102*, it should activate our app.

This is a way of ensuring that once we get the authorization code back, it goes to our app rather than the browser.

Finally, we have the *code_challenge* parameters. I mentioned the code challenge earlier when I introduced the PKCE mechanism. The application provides this code to the authorization server, which will tie the authorization code to this challenge. We'll see how this comes together shortly.

The *code_challenge_method* is just the implementation details of the algorithm used to generate the challenge.



```
(1) GET https://flosser.auth0.com/authorize?
?audience=https://flosser.com/api/
&response_type=code
&scope=openid profile offline_access read:appointments
&redirect_uri= id102://flosser.auth0.com/cb
&code_challenge=KuijW7HRw10_ie7qeVmyyHrkcn-d
&code_challenge_mode=S256
```

```
(2) 302 HTTP/2.0
location: id102://flosser.auth0.com/cb/?code=35b0NnPFoolksUQp
x-auth0-requestid: b86850400a490f385619
set-cookie: auth0=s%3AT_[...]8mv7wE; Path=/; Expires=Fri, 14 Dec 2018 13:40:27 GMT; HttpOnly; Secure
```

```
(3) GET id102://flosser.auth0.com/cb/?code=35b0NnPFeolksUQp HTTP/1.1
?code=35b0NnPFeolksUQp
```

```
(4) POST https://flosser.auth0.com/oauth/token HTTP/2.0

audience=https%3A%2F%2Fflosser.com%2Fapi%2F
&client_id=IrbblpeZJRMewo8suTx0PcmHlporj6yZ
&redirect_uri=id102%3A%2F%2Fflosser.auth0.com%2Fcb
&grant_type=authorization_code
&code_verifier=NmFCZzFkSHI1a0R0M3NnSkVuY1FZNzRBOEE1REJ2VE0
&code=35b0NnPFeolksUQp
```

```
(5) 200 HTTP/2.0
{
  "access_token": "ey [...]",
  "refresh_token": "ZjOmJYYjkluu3pHZe_IdT5OTD89LijNACC4_sgw9cGDwN",
  "id_token": "eyJ0 [..] ",
  "scope": "openid profile read:appointments offline_access",
  "expires_in": 86400,
  "token_type": "Bearer"
}
```

```
(6) GET https://flosser.com/api HTTP/2.0
authorization: Bearer eyJ0eXA[..]TOg
```

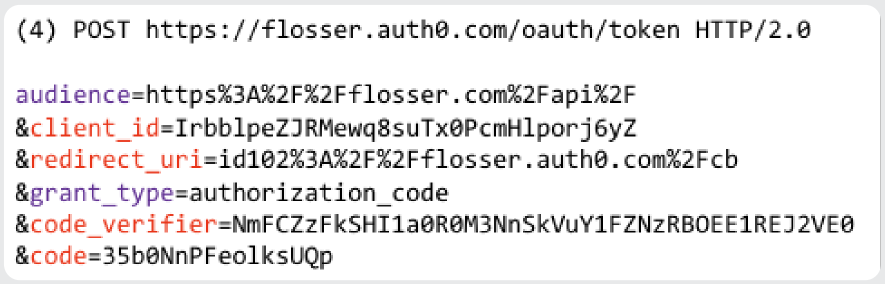
Figure 5.2

2. **Authorization Response**
   So now you'll have all the usual back-and-forth steps you expect to complete the authentication process, including the consent step and whatever MFA might come into play. What's important from a protocol perspective is that we get back our usual response, in which you can see the 302 HTTP status code on the location that we specified.

   This is our redirect URI, and we are getting an authorization code, exactly what we asked for. We also have the usual *set-cookie* as a result of successful authentication.

3. **Redirect to the Application**
   Now comes the original part, when the browser executes the 302 redirection. Since the protocol handle is *id102*, this is actually a communication within the device. The browser gives back control to the application passing the code. In this step, even if someone is in the middle and steals the code, it doesn't matter because they can't use it. I'll show you why shortly.

4. **Exchange the Authorization Code**
   Now that we have the authorization code in our application, we can turn around and finally go to the token endpoint. It's a classic redemption flow with the only caveat that we don't provide a secret. Remember that a public client does not have a secret.



```
(4) POST https://flosser.auth0.com/oauth/token HTTP/2.0

audience=https%3A%2F%2Fflosser.com%2Fapi%2F
&client_id=IrbblpeZJRMewq8suTx0PcmHlporj6yZ
&redirect_uri=id102%3A%2F%2Fflosser.auth0.com%2Fcb
&grant_type=authorization_code
&code_verifier=NmFCZzFkSHI1a0R0M3NnSkVuY1FZNzRBOEE1REJ2VE0
&code=35b0NnPFeolksUQp
```

Figure 5.3

It's the usual POST to the token endpoint. We have the *audience* parameter and the *client_id*. We don't have a secret but have the *redirect_uri* parameter, which, again, we specify for security purposes. As you can see, it's still the one with the *id102* protocol handle.

The *grant_type* is *authorization_code*, and we provide the authorization code as the last parameter in the URL.

A new element here is the *code_verifier*, which proves to the authorization server that our application is still the same requester of the authorization code. Anyone who stole the token while it was passing from the system browser to the app would not be able to produce this code. That's pretty handy.

5. **Get the Tokens**
   As a response, we get back our usual access token, refresh token, ID token, the list of consented scopes, the *expires_in* value - because we can't look inside the access token - and the *token_type:* all ordinary administration. Pretty straightforward.

6. **Call the API**
   Now, our application has the tokens that allow it to work as expected. It has the ID token with the user claims and the access token to call the API in the same way we learned in Chapter 4.

The Problem with Refresh Tokens

If our native application receives a refresh token, it will be using it in the same way as confidential clients. However, unlike confidential clients, our application doesn't provide a secret because it doesn't have any. Of course, this is a problem because refresh tokens from public clients can just be used as-is. So those are little magic things that will keep minting tokens without any need for doing any excess stuff.

Before I go too deep into this, let me show you how the Refresh Token grant works in this case. Say that I want to get a new access token. I send the following message to the token endpoint:

```
(1)   POST https://flosser.auth0.com/oauth/token HTTP/1.1
      client_id=xHMI55zgwY0PnaztfSQflbFAwxxHUM8_
     &refresh_token=zisHo7hCAqOQzGDST1GmF_E1OEV4OVxS5GYOzIBh_uZrA
     &grant_type=refresh_token
     &redirect_uri=id102://flosser.auth0.com/cb
```

Figure 5.4

There is nothing notable here. It's exactly the same stuff that we had earlier, with the difference that we don't have a secret. So here, there is no code, no PKCE. It's just the same redemption of a refresh token but without a secret.

This is clearly a problem, and in fact, lots of people are very nervous about it, although it's the mainstream. That's what everyone does. So, as an industry, we are looking to other solutions that do not necessarily entail creating a confidential client on the native devices. It's more about finding ways to bind the tokens to the channels used when receiving them.

Token Binding

There have been a couple of efforts in the industry. One is called Token Binding, and it's a set of specifications used to extend the HTTPS stack and browsers' ability to surface properties of HTTP stacks that can be embedded in tokens.

When you use tokens, the authorization server and the resource server can actually verify that the tokens are being used in the same channel they were requested for. If this doesn't happen, that basically means that someone stole that token and they are trying to use it from elsewhere. So you can prevent this from happening by refusing to serve the request.

This was a good idea, but it required many planets to align. And the planets didn't align: Apple didn't announce support for this; Chrome had support for it, but then Google announced it would stop supporting Token Binding. In the end, the specification was retired.

Mutual TLS Authentication

Another specification is the alternative to the Token Binding flow: Mutual TLS Client Authentication. This specification has the great advantage of using capabilities that are already present on browsers and operating systems, such as client certificates.

An authorization server can require the application to use a client certificate to authenticate and get the tokens. This authentication occurs at the network (TLS) level. Then, the same certificate can be required when you use the obtained tokens. As a result, if you take one of those tokens and try to use it from a device that doesn't have that certificate, you won't be able to.

Application Level Demonstration of Proof of Possession (DPoP)

Given that Token Binding did not become a generally available mechanism, the OAuth 2.0 Demonstrating Proof of Possession (DPoP), now RFC 9449, was introduced. Given the application level constraints, this specification allows clients capable of generating non-extractable asymmetric keys to demonstrate their proof of possession, which in turn allows the authorization server to bind tokens to them. Similarly to Mutual TLS, the tokens bound with DPoP cannot be used unless also having access to the keys they're bound to. This mechanism is not as strong as Mutual TLS but does not come with deployment hardships and browser UX hurdles stemming from TLS Certificate system popups that can plague Mutual TLS setups.

A Final Note

A final thing I want to mention about refresh tokens in the context of native clients is to reinforce what I said earlier. The refresh tokens are the artifact that tracks your ability to get tokens. They help you give the user a smooth session experience without interruptions and you typically have to follow all the session management steps to also ensure security. For example, when you want to terminate a session, you typically want to delete the refresh tokens from your cache as well.

The Resource Owner Grant

Let's talk about another controversial grant you might encounter when you want to create native applications: the *Resource Owner Password* grant. This is pretty much what you can think of: a grant that allows you to take a user's username and password and programmatically post them to the authorization server to get a token. Crude but effective.

The Bad Part

In the context of delegated authorization, the direct usage of credentials is dangerous. It doesn't give you any of the expressive power you normally have with all the mechanisms that we visited so far. In general, it just encourages the user to do the wrong thing. It trains users to enter their credentials in interfaces other than the ones that own those credentials. Typically, people want to do this when they want to have their own UI instead of a web page.

But in general, whenever you trade with raw credentials, especially in the native application space, we use an external browser instead of an embedded browser. Every time you use raw credentials, you put yourself in potential jeopardy. In fact, the Resource Owner Password grant is deprecated in the upcoming OAuth 2.1 specification.

Apart from the security aspect, here is a partial list of shortcomings when you directly use a username and password:

- **You cannot prompt for consent**
  So, any resource gated to user consent cannot be used unless you take prior steps to register with consent. This is bad both from the mechanics and the optics: the user is not aware of what's going on or how those credentials will be used.
- **You cannot do multi-factor authentication**
  unless you embed the capability of doing so in your client application. That's what happened years ago before we introduced the use of a browser in the context of authentication. I assure you, it wasn't fun at all. Whenever you want to change even the tiniest thing, you have to redistribute your code to all your clients, some of which might only occasionally be connected. So you'll have people who call that up for the first time and discover it no longer works for years. It's really bad.
- **You can't do step-up authentication**
  If you have different resources that require different levels of authentication, you can't really do that. You can only send a username and password at login time.
- **You can't use multiple identity providers**
  Consider when, during the authentication ceremony, you get prompted with a list of identity providers from which you can choose. So maybe there is a button "Sign-in with Facebook" and a button "Sign-in with Google" or a field for entering your corporate email that will redirect you to your corporate identity provider. Your application can't do this if you are using the Resource Owner Password grant. Even if there was a way for you to expand or contract this list, there is no way for you to connect to the providers that don't allow you to programmatically send credentials, which is the case for most of the serious ones.

- Finally, **there is no single sign-on.** If I have a cookie somewhere that says I'm already authenticated with Facebook, and I click a little button that says, "Sign-in with Facebook", I'll just be bounced back and forth. I will take advantage of the fact that I already have a session-tracking cookie and will not have to enter any credentials.

As you know, having native fields for usernames and passwords is very bad in the OAuth and OIDC context. However, as an Identity professional, you will face many arguments in favor of the Resource Owner Password flow and against other browser-based flows.

You must expect the non-initiated to ask for the Password flow often and with emphasis, mainly because it's simple. They might feel overwhelmed by all these million parts, browsers popping out, redirects, and so on. Simplicity is a tempting aspect of this flow.

In addition, you can often hear concerns about control over the user experience. Luckily, in Auth0, we don't have this problem because developers have control over the user experience of the login page.

People might have concerns about performance because of redirects. They might think about redirects and say, "Wow, it's going to be a hit in performance". Usually, a good idea is to actually test and show people that this is not the case. It's normally pretty straightforward.

But there is at least one case in which, in my career, I never managed to find a way to avoid this flow: in pure legacy scenarios.

Imagine that you have an application that already gathers usernames and passwords, and you cannot touch its code. Assume that changing the code for the authentication part is very difficult. That codebase may be old; maybe the person owning the code is no longer with the company.

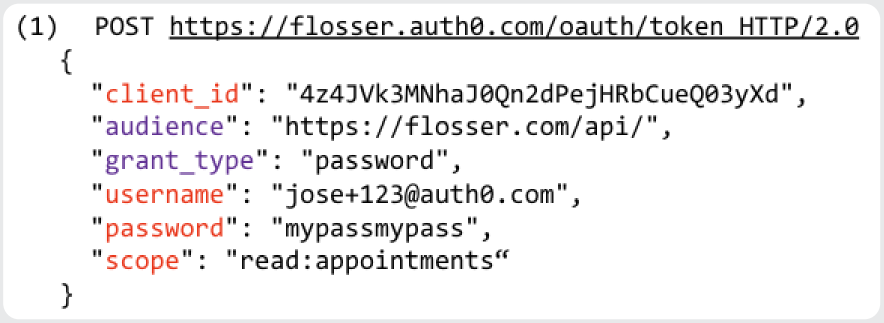
Or think of scenarios where you have a script with a connection string. If the connection string only has a username and password, and you should use this script as is, then you need to bridge some of the gap.

So, for pure legacy scenarios and cases where there is a plan for moving forward and moving off of sheer username and password, I usually tend not to complain too much if people ask me for a review for that scenario. But it's the only scenario. For all the other scenarios, I will always insist on using some finesse because this flow is just a problem waiting to happen.

The Flow Description

Now, after all those dire warnings, I will show you how the Resource Owner Password flow works anyway so that if you have to do this, you know what to deal with. Following the diagram in Figure 5.6, we have a scenario with an application and an API. The diagram uses the icon for the desktop app because, as I said, this flow is only remotely acceptable for legacy applications, and you cannot have legacy applications on the phone since the phone is just too new.

1. **Send the user credentials**
   As I mentioned, you have some mechanism for gathering usernames and passwords, probably some old-fashioned mechanism. Then you'll just turn around and send those credentials to the token endpoint, and here is how the message looks:



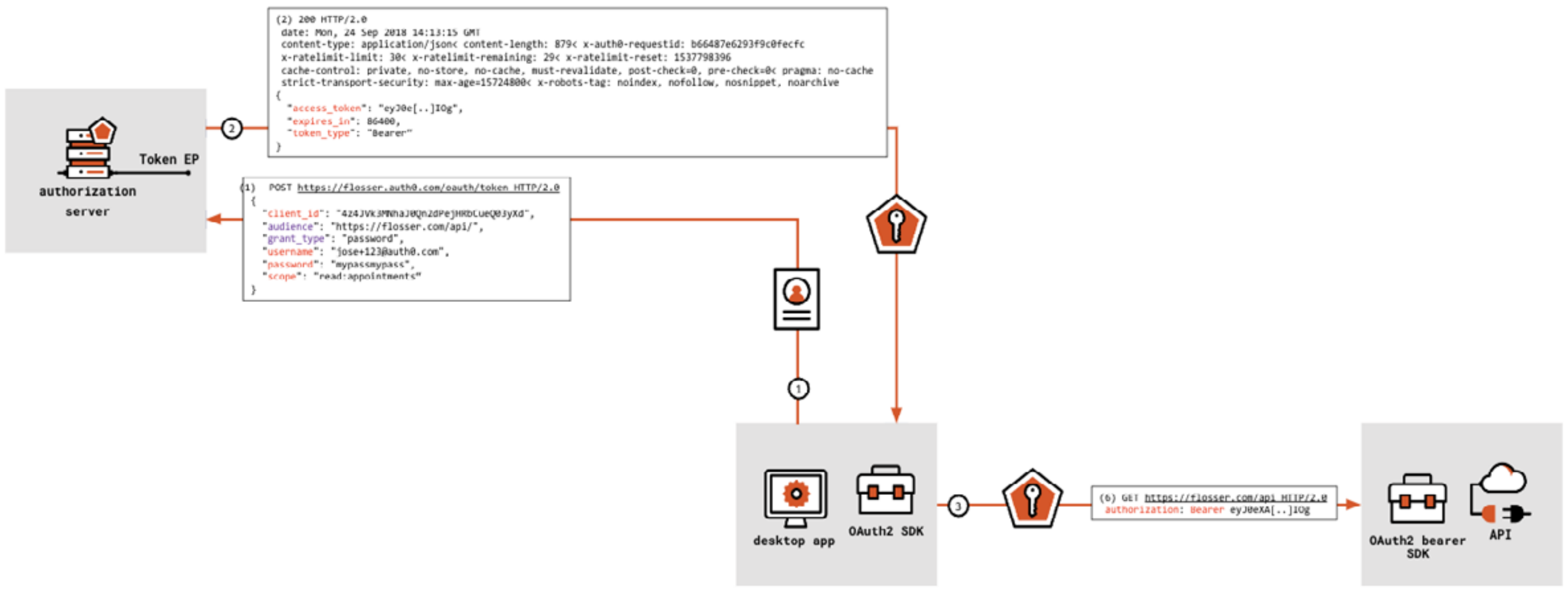
```
(1)  POST https://flosser.auth0.com/oauth/token HTTP/2.0
   {
     "client_id": "4z4JVk3MNhaJ0Qn2dPejHRbCueQ03yXd",
     "audience": "https://flosser.com/api/",
     "grant_type": "password",
     "username": "jose+123@auth0.com",
     "password": "mypassmypass",
     "scope": "read:appointments“
   }
```

Figure 5.5

   We have the usual *client_id* and *audience* parameters. There is the *scope* requested by the app and the *grant_type* with its value set to simply "password". Then, there is the user's *username* and *password*. Remember, we are calling the token endpoint.

2. **Get the response**
   As a response, the app gets the usual data: the access token, the *expires_in* value, and the *token_type*. That's it.

3. **Call the API**
   Of course, I can grab the token and use it to call the API. So, it's just your basic normal flow.



```
(2) 200 HTTP/2.0
 date: Mon, 24 Sep 2018 14:13:15 GMT
 content-type: application/json< content-length: 879< x-auth0-requestid: b66487e6293f9c0fecfc
 x-ratelimit-limit: 30< x-ratelimit-remaining: 29< x-ratelimit-reset: 1537798396
 cache-control: private, no-store, no-cache, must-revalidate, post-check=0, pre-check=0< pragma: no-cache
 strict-transport-security: max-age=15724800< x-robots-tag: noindex, nofollow, nosnippet, noarchive
{
  "access_token": "eyJ0e[..]IOg",
  "expires_in": 86400,
  "token_type": "Bearer"
}
```

```
(1)  POST https://flosser.auth0.com/oauth/token HTTP/2.0
 {
   "client_id": "424JVk3MNhaJ0Qn2dPejHRbCueQ03yXd",
   "audience": "https://flosser.com/api/",
   "grant_type": "password",
   "username": "jose+123@auth0.com",
   "password": "mypassmypass",
   "scope": "read:appointments"
 }
```

```
(6) GET https://flosser.com/api HTTP/2.0
 authorization: Bearer eyJ0eXA[..]IOg
```

Figure 5.6

Other Grants for Native Apps

There are other grants somehow related to native applications that I'm not covering in this book. However, they deserve at least a mention.

The Device Authorization Grant

One of these grants is the *Device Authorization* grant. It applies to scenarios involving devices like smart TVs, media consoles, or other IoT devices. These are devices with a limited display and no browser. A server in a server farm is another example where this grant can be applied. Typically, you have a CLI that runs on a server with no graphical capabilities, but you still want to call APIs. You learned that in a delegated authorization scenario, you need a browser. How can you authenticate and authorize an application that runs in an environment without a browser?

For this purpose, the Device Authorization flow uses a trick. It shows you a code in the text-based interface and instructs you to pull out a different device with a browser and navigate to a given address. Once there, enter that code on the page loaded in your browser. Once you do this, you'll be driven through the classic experience you need to do for authentication: MFA, consent, and anything else the authorization server deems appropriate.

The client running on the text-only device will constantly poll the authorization server. As soon as you give consent, this polling will be successful, and the application will receive the tokens it needs. It's really straightforward and super handy.

The Token Exchange Grant

The second flow I'd like to mention is the Token Exchange grant. This grant allows you to exchange a token for another. In a nutshell, consider a scenario where a client calls an API, and this API needs to turn around and call another API, carrying forth the identity of the original caller.

The first API can use the incoming access token as a grant to exchange it for an access token to call the other API.

Extension Grants

The Token Exchange grant is actually a specific implementation of the extension grant mechanism provided by the OAuth 2.0 specification. This mechanism allows you to define a custom grant when the existing standard grants do not apply to a specific scenario.

Another example of an extension grant type is the SAML Profile for OAuth 2.0. This grant is similar to the Token Exchange grant: the application already has a SAML token obtained using a legacy scenario and wants to turn it into an OAuth 2.0 access token. While in the case of the Token Exchange grant, you remain in the same OAuth context, the SAML profile grant enables interoperability between different authorization contexts.

These are the grants that you might experience in the context of native clients. However, you can also use them for confidential clients, especially the Token Exchange grant.

So that's it for native clients, i.e., desktop and mobile clients. Next, we'll explore the world of Single-Page Applications.

Chapter 6

Single Page Applications

What are Single Page Applications or SPAs? I am sure that most of you know what we are talking about, but let's try to better describe this type of web application.

The Nature of Single Page Applications

You use information-dense web-based UIs like Gmail or Outlook Web Access and similar. These are web pages that present a lot of information at the same time. The natural interaction that the user has with this kind of application requires updating just parts of the interface. If you would be implementing that interaction model using classic postback, you'd be doing a lot of useless work.

Imagine a typical layout with a list of messages on the left and a panel showing the content of a selected message on the right. Whenever I click a different message, all I want is for the selection to move to where I clicked and for the content of the preview panel to update itself.

That's it.

If I were using a classic postback-based web application, I'd go back to the server and ask for the entire page again: the list of messages, all the visual elements around it, and icons. Sure, I can do caching, and I have tricks that can make things better, but that would be a lot of traffic and also pretty bad performance.

In most cases, all I do with a Single Page Application is get a single page from the server at the beginning. This page contains the basic visual elements and, together with that, a lot of JavaScript, which can reach out to the server and ask only for the data. JavaScript takes this data and uses it to reflect changes in the UI. It programmatically injects the new data so that it gets displayed. So I don't have to get an entire page every time, just the needed part. That's super handy. Without that, we would not have the modern web experiences that we enjoy today.

Security Challenges of Single Page Applications

From a security perspective, this is an interesting conundrum. It's a web page that lives inside the browser. It's subject to all the classic attacks to which something inside a browser is subject. Since it runs inside a browser, it's isolated from the device in itself. At the same time, the interaction I just described is largely based on the client reaching out and calling an API.

Sounds familiar? Yes. That's pretty much the same stuff we have seen with a native client. Now we have this interesting challenge: How do we secure this thing? How do we deal with identity in this case?

We typically treat this type of application somewhat as a native client. It gets tokens and uses them to call the protected API. However, we'll see how this approach opens up some challenges that are not present in the native client case.

Where we keep the APIs makes a huge difference. If the web API is in the same domain from where we are getting the single page of the application, then you can think of securing the traffic in whatever way we have done in the past for websites. So cookies are viable, but the moment you need to start making API calls outside of your domain, then cookies are no longer viable because they are tied to a domain. Your browser can't just attach cookies to JavaScript calls to other domains. So, the token-based approach is the most generic, the one you can use in every situation. It's also the one with the most moving parts and, consequently, the most brittle. So we'll see what that means in terms of trade-offs.

Single Page Applications and the Implicit Grant

In the early days of OAuth, the traditional way to secure Single Page Applications was through the Implicit Grant. Now, you have heard the word "implicit" earlier in the context of signing in for web applications, specifically in the particular style of the *Implicit flow with Form Post*. From the OAuth point of view, a grant is *implicit* when you are getting a token directly from the authorization endpoint instead of having to trade something with the token endpoint. That's the formal definition of *implicit*.

Interestingly, when you talk about implicit, the most salient scenario that people will think of, like the classic use, the default meaning in literature, is what I'm going to explain right now in relation to Single Page Applications, that is to say, using the *Implicit* grant and delivering the tokens in a URI fragment (the part that comes after the pound sign (#) symbol). This scenario is fraught with issues, but other scenarios are perfectly fine, such as the one in which we use the Form Post and follow all of the necessary ID token validation steps.

The Implicit Grant used for Single Page apps gives the entire *Implicit Grant* family a bad name. In fact, the bad name is only well deserved in

the context of Single Page apps, whereas all the things we did earlier for the web sign-on on the front channel are perfectly fine, assuming the application is only getting an ID token and validating it properly.
So don't be worried, and be prepared to explain every time you use the *Implicit Grant with Form Post* that it's not that *Implicit*.

Let me give you a bit more concrete indications about what I mean by a bad name. As with any *Implicit Grant*, we use web page redirects to ask and obtain tokens directly from the authorization endpoint. In the case of the *Implicit Grant with Form Post*, the token travels in the body of the page. In the case of *Implicit Grant with Fragment*, the token is returned as a fragment element in the redirect URL.

So, having the token in the body of the page has slightly fewer security risks than placing it in the URL. The thing is that although no expert in the identity space ever liked this flow, it was the only game in town, since browsers could not support cross-domain POST requests at the time. So pretty much all the Single Page Applications used the Implicit flow and the fragment approach. We'll see that things changed pretty fast.

The complications here are, to some extent, intrinsic to the fact that we live inside a browser, i.e., an open platform, and the more open a platform is, the bigger the attack surface. There are all sorts of ways in which things can go wrong. For example, if you save your tokens in your local storage, in the case of a cross-site scripting attack that dumps your entire local storage content, your tokens are compromised. If you receive the tokens within your URL, this URL will end up in the browser history but also in the referral headers. So, there are more ways of leaking those tokens.

To mitigate the effect of a leaked token, we can reduce its validity time. This entails the need to renew it, but using an artifact that's powerful as a refresh token within the browser, as we do for native clients, can also be a problem. A native client living on a local resource can have some degree of protection, but we cannot afford to have it within a browser.
So, it is not advisable to use refresh tokens within the browser unless you do token rotation.

In conclusion, the Implicit flow has a number of complications, which ultimately led the OAuth working group to suggest that we abandon it and do something else.

SPAs and the Authorization Code with PKCE Flow

As I mentioned, the OAuth working group found that the risks we have when using the Implicit Grant to get access tokens outweigh its convenience. The alternatives to this flow rely on features that weren't available across the board when the Implicit flow was first devised, such as *Cross-Origin Resource Sharing* (CORS), for example.

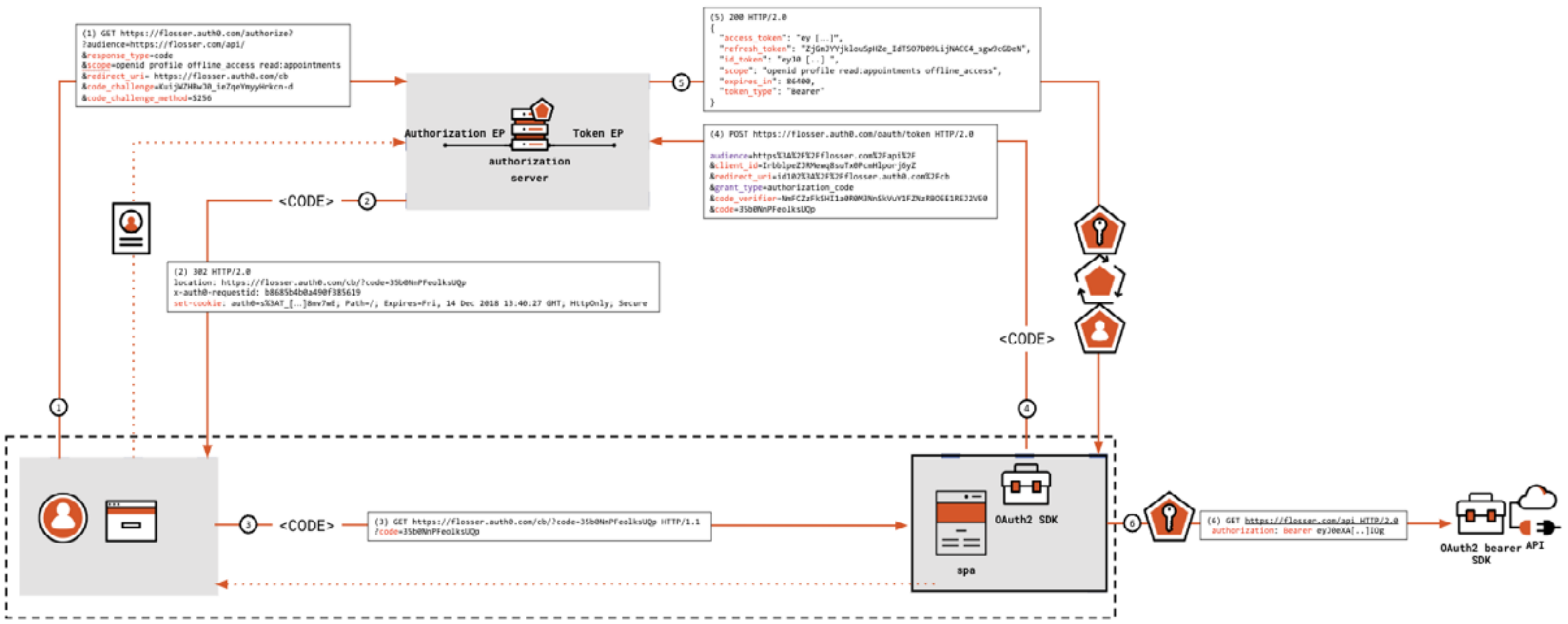
The idea is that we can actually use the exact same flow that I described for native clients for Single Page Applications as well. In this case, you use JavaScript to implement the *Authorization Code flow with PKCE*. Take a look at Figure 6.1, which shows the exact same flow of a native application.

As usual, you use the browser to get the authorization code from the authorization endpoint. Once you have this code, you can use JavaScript to hit the token endpoint, and then you will get an access token from that channel. That channel will not expose the access token to the browser history and will not expose it in headers. So, it's way easier to protect than all the mess you must do when you use redirects to obtain tokens directly in the URI.

Note that here we can use the classic HTTPS scheme instead of the custom protocol handle we used for native applications.

This flow can also use refresh tokens as long as you use one of the mechanisms I suggested earlier to protect refresh tokens.

One is the refresh token rotation: when you use a refresh token, it's no longer valid. You get a new refresh token to use from that moment on. This mechanism is deemed enough to protect refresh tokens in the browser. The alternative mechanism is to use a standard constraint mechanism like DPoP, that I described at the end of the chapter about native clients.



```
(1) GET https://flosser.auth0.com/authorize?
?audience=https://flosser.com/api/
&response_type=code
&scope=openid profile offline_access read:appointments
&redirect_uri= https://flosser.auth0.com/cb
&code_challenge=KuijW7HBw30_ie7qeYmyyWrkcn-d
&code_challenge_method=S256
```

```
(2) 302 HTTP/2.0
location: https://flosser.auth0.com/cb/?code=35b0NnPFeolksUQp
x-auth0-requestid: b8685b4b0a490f385619
set-cookie: auth0=s%3AT_[...]8mv7wE, Path=/, Expires=Fri, 14 Dec 2018 13.40.27 GMT, HttpOnly, Secure
```

```
(3) GET https://flosser.auth0.com/cb/?code=35b0NnPFeolksUQp HTTP/1.1
/code=35b0NnPFeolksUQp
```

```
(4) POST https://flosser.auth0.com/oauth/token HTTP/2.0

audience=https%3A%2F%2Fflosser.com%2Fapi%2F
&client_id=IrbblpeZJRMemq8suTx0PcmHlpurj6yZ
&redirect_uri=id10%3A%2F%2Fflosser.auth0.com%2Fcb
&grant_type=authorization_code
&code_verifier=NmFCZzFkSHI1a0R0M3NnSkVuY1FZNzRBOEE1REJ2VE0
&code=35b0NnPFeolksUQp
```

```
(5) 200 HTTP/2.0
{
  "access_token": "ey [...]",
  "refresh_token": "ZjGnJYYjklouSpHZe_IdTSO7D09LijNACC4_sgw0cGDeN",
  "id_token": "ey10 [...] ",
  "scope": "openid profile read:appointments offline_access",
  "expires_in": 86400,
  "token_type": "Bearer"
}
```

```
(6) GET https://flosser.com/api HTTP/2.0
authorization: Bearer eyJ0eXA[..]10g
```

Figure 6.1

SPAs with a Backend

We use tokens to secure Single Page Applications. This will allow us to call APIs no matter where the APIs live, whether it's our backend or elsewhere. We might have no backend at all and call APIs exposed by others or by ourselves but on a different domain, such as in a serverless environment. However, having a backend of your own is pretty common. For example, if you're serving your SPA from some kind of active page technology like Node.js or ASP.NET, then you have a backend.

Let’s explore how we can take advantage of having a backend available to our SPA.

SPA and API on the same domain

If you expose your API for the exclusive use of your presentation layer in JavaScript, then technically, nothing prevents you from using the same technique we use for web sign-on. From a browser perspective, it doesn't matter whether the thing you are trying to access on your web server is a page meant to be seen by a human or if it's an API endpoint used to retrieve data. It's just an HTTP verb hitting a certain domain. If a cookie for the server's domain exists on the browser, it will just attach it and send it along with the request. On the backend, if a middleware sits in front of those routes, it also doesn't care whether the request is trying to access a page or an API endpoint. As a result, as long as we are in that particular scenario, the backend does the same stuff I described for the web sign-on at the very beginning. It does that when the SPA is requested the very first time, too. From that moment on, you can just use cookies to protect your API calls. The same middleware that triggered this sign-in on the first page will enforce the presence of a cookie, and you'll use it for authenticating.

Having the API in the same domain as the SPA allows me to simplify things. I'll probably do the *Implicit* flow with Form Post to get an ID token and exchange the token for a cookie. Alternatively, if I want to do server-side flows, I can do the *Authorization Code* flow and redeem the code for an ID token. All the techniques we described for web sign-on can be applied in this context.

I only have two challenges to think of in this scenario.
Consider that the authentication flow relies on redirects. Say that your

cookie expires at a certain point, and you are making an HTTP request from your SPA. You will get back a 302 HTTP status code because, from the middleware perspective, you are trying to access this page and are not authenticated. So, the middleware will send you to the authorization endpoint. But an HTTP request from a SPA doesn't really know what to do with a 302 status code. You need an error management logic that intercepts this 302 status code and shows the user some affordance, like, "Click here to reauthenticate."

I don't recommend automatically redirecting the user because if they are in the middle of filling out a form and you ship them away, you are not offering a good user experience. You could show a popup window, but popups are controversial because sometimes they are blocked. So, in general, my advice is to show a little toaster that says you've got to re-authenticate.

The other challenge is that your JavaScript-based application will want to access the user information sooner or later. To do so, you expect your JavaScript to be able to go somewhere and find out the user's first name, email, etc. All stuff that you would normally get if you'd get the ID token. But in this scenario, the ID token was received by your backend, and your SPA works with cookies. Cookies are purposefully opaque to the client, so you cannot extract information from there.

You need to add a route to your API that allows the JavaScript application to query the tokens' content and, in general, obtain the user information that the JavaScript layer requires.

The Token-Mediating Backend Pattern

Having a backend for our SPA allows us to do all the flows we have seen for calling an API from a web page. For example, as a confidential client, your backend can obtain tokens using the *Authorization Code* flow. Can we leverage this scenario to call third-party APIs, i.e., APIs that require an access token? The answer is yes, you can do it. One way to achieve this is to use the Token-Mediating Backend pattern.

You have your JavaScript application that wants to call an API and your backend. Your users have an interactive web flow, which will lead to performing the classic *Authorization Code* flow, redeeming the code, and obtaining an access token and refresh token. Then you can just take that access token and send it back to your JavaScript app. Your SPA can use that token to call APIs from JavaScript.

As in the previous case, we are not relying on any part of the OAuth flow happening on the client. The client doesn't really have any code for obtaining tokens.

This approach leverages the security profile of a confidential client - the backend - to get tokens from the authorization server. This relieves your SPA from implementing the authorization flow with all the potential issues. However, this comes at the cost of a significant complication on the server side, which, in addition to obtaining the tokens, must also take care of making the access token available to the client.

Notice that only the access token is sent back to the SPA. The refresh token is kept on the backend and associated with the user's session. When the API rejects the access token, the SPA contacts the backend to request a new access token. Then the backend uses the refresh token associated with the current user to request a new access token and send it back to the SPA. In other words, the SPA will never have to directly deal with refresh tokens.

By delegating obtaining tokens to the backend, you reduce the attack surface of the SPA. However, the JS application still remains exposed to attacks that allow an attacker to steal the access token and call the remote API. In this regard, following best practices to mitigate these risks is advisable, such as avoiding storing tokens locally in the browser. For more details on the pattern and its security considerations, see the OAuth 2.0 specs.

The Backend for Frontend Pattern

There is another way to take advantage of the backend's potential and lighten a SPA's security burden: delegating the responsibility of interacting with the authorization server and managing the tokens entirely to the backend. The pattern we are going to explore is known as Backend for Frontend (BFF) and is essentially based on attributing the role of the intermediary to the backend both towards the authorization server and third-party APIs.

The backend takes care of interacting with the authorization server as a confidential client: it redirects the user to the authorization endpoint for authentication, obtains the authorization code, exchanges it for ID, access, and refresh tokens, and behaves as a normal confidential client.

As in the previous cases, the backend tracks the user's authenticated session with a cookie, but unlike the *Token-Mediating Backend*, it does not forward the tokens to the JavaScript application. It stores them on the server and exposes some endpoints with which the SPA can interact for all its needs. For example, it exposes an endpoint that provides user profile data, which the backend extracts from the ID token. Furthermore, the backend acts as a proxy between the SPA and the API. All SPA calls directed to the API pass through the backend, which exposes one or more endpoints for this purpose. When the backend receives a request to these endpoints, it checks that it contains the session cookie and forwards the request to the API after including the access token. Once the response is received from the API, the backend forwards it to the SPA, and that's it.

This pattern offers the security of a confidential client to a public client like the SPA. The JavaScript application never touches the tokens, so there is no risk of them being stolen at the SPA level. As the specs say, *“Because of the nature of the BFF architecture pattern, it offers strong security guarantees. Using a BFF also ensures that the application's attack surface does not increase by using OAuth. The only viable attack pattern is hijacking the client application in the user's browser, a problem inherent to web applications.”*

The price to pay for improving the security of the SPA with this architecture is the increased complexity of implementing the proxy mechanism between the JavaScript application calls and the remote API. This, among other things, affects call performance, which must, therefore, be taken into account when considering adopting this pattern.

To learn more about the threats, attack consequences, security considerations, and best practices for SPAs, you can check out the OAuth 2.0 for Browser-Based Applications document.

Conclusion

We have reached the end of this book, but this is not the end of the journey for an aspiring Identity professional. The topics covered in this book are just the foundation of OAuth 2.0 and OpenID Connect-based Identity.

We have not covered several topics that are part of the OAuth framework. We have simply mentioned some, such as the Device Authorization and the Token Exchange grant or JWT Profile for Access Tokens; we have described other concepts at a high level without going into detail, as we did for DPoP and MTLS; and we haven't even mentioned other topics, such as Dynamic Client Registration, Token Revocation, and many other extensions.

Furthermore, recently, we have seen a demand for ever greater security, especially in areas such as finance, insurance, healthcare, and utilities. In these areas, security and privacy are of fundamental importance. For this reason, the OAuth and OIDC community has finalized a series of extensions that strengthen these protocols for use in critical scenarios. Part of these efforts feed into the FAPI specifications.

However, beyond the mere concepts explained here, I hope you got the core spirit of OAuth and OpenID Connect, the motivation behind their birth, the reason why things are how they are, and the motivation for their evolution to fulfill the identity security needs of the industry. With this foundation, I hope you can design and implement more robust applications and choose the most appropriate solution for the needs of your architectural scenario.