OAuth2 and OpenID Connect: The Professional Guide - *Beta*

by Vittorio Bertocci
curated by Andrea Chiarelli

Auth0

OAuth2 and OpenID Connect: The Professional Guide - *Beta*

by Vittorio Bertocci
curated by Andrea Chiarelli

Auth0

Introduction

This e-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 OAuth2 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 OAuth2 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 can do for our 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 these complex is the Cartesian product of all the factors that come into play to determine what you have to do for having 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 in 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 that 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 that 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 that 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 that you wanted to access, the places from where you are extracting information. So, this picture might look simple, but it's all but.

Now, what can Auth0 do for you to make this a bit more manageable? We offer many different things but, in particular, the most salient component of our offering is our service. 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 would have 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 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
- ✓ **OAuth2**, 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'd be 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 that 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 which is 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 that the user 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 which 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 that we want to achieve, we actually have a completely or nearly completely different set of identities. 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 in 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, 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:

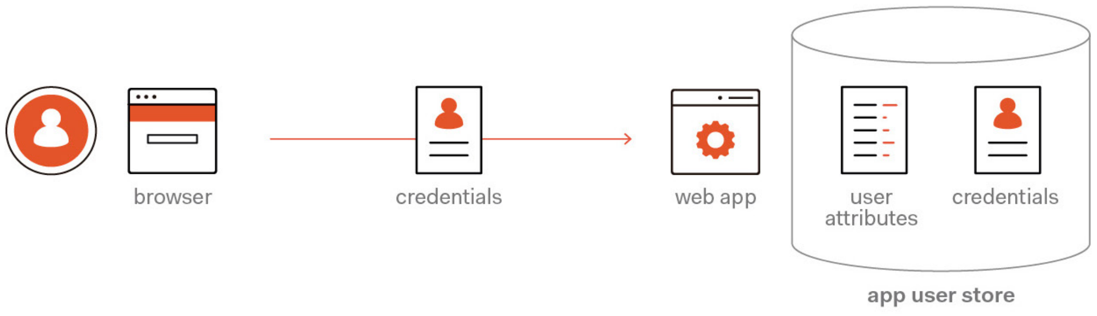


Figure 1.2

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 his or her 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 which knew nothing about each other, and which 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 in 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 for buying large amounts of items just for getting 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.

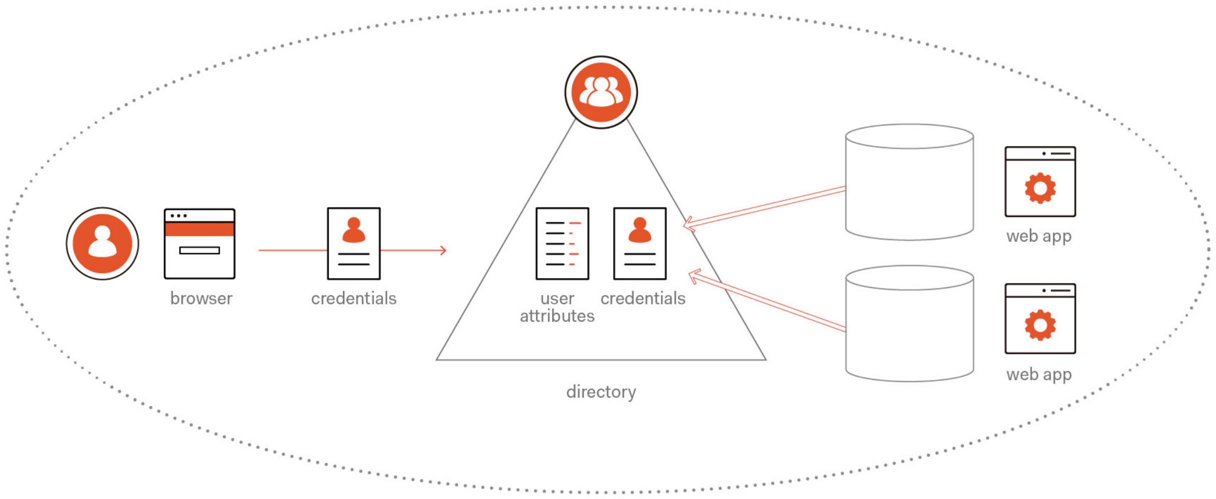


Figure 1.3

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 to 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 in which 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 that 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 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, are harder to reflect in distributed systems not centrally managed. Plus, imagine the complexity of having this company, which may be a reseller for many other companies, but needs to duplicate somewhat 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 that 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 is that the big guys of that time, Sun, IBM and similar, 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 which facades my actual resource with some software which is 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), and 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 in 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 that you're breathing because there is only one place that can perform authentication duties in the entire network.

Now, look at this particular scenario:

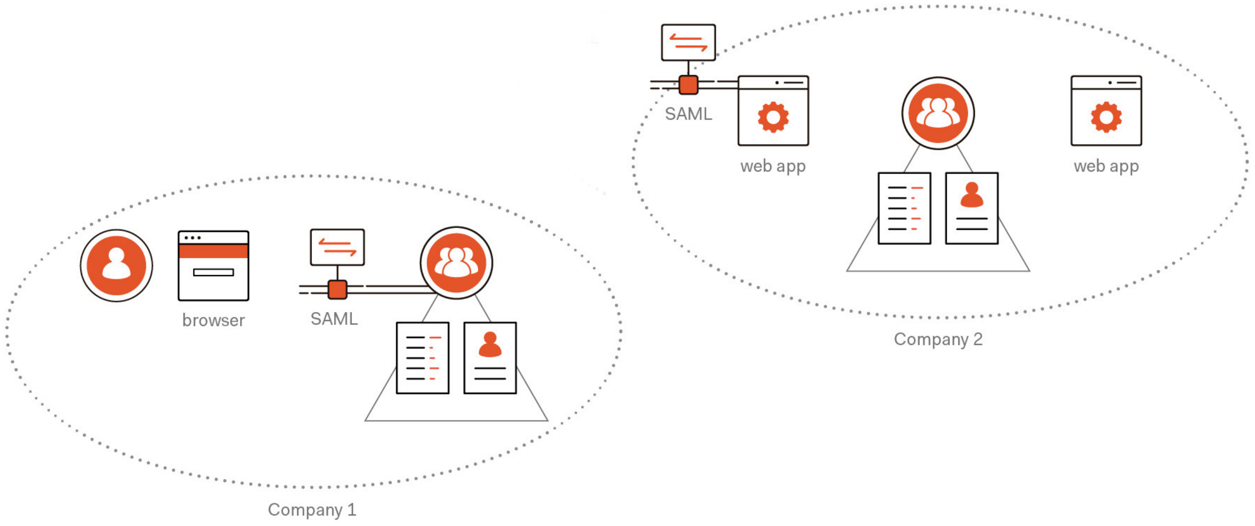


Figure 1.4

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, 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:

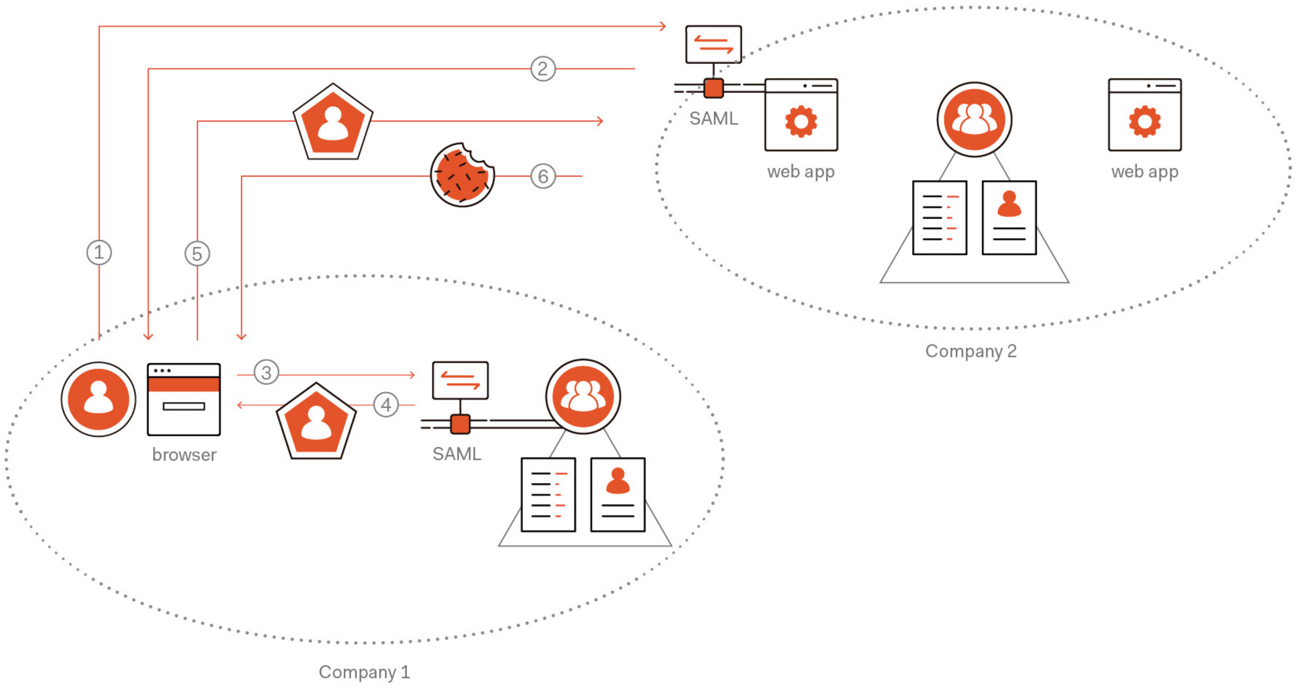


Figure 1.5

In the first leg of the diagram, the user points the browser to the application and attempts to GETa 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 establishes that the resource is one of the resources that have been recorded and approved. Because of those positive checks, the IdP issues to the user what we call a **security token (4)**. A security token is an artifact, a bunch of bits, which is used to carry a tangible proof that the user 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 anything 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 that was 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 just 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 a 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 does indeed possess 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 this potential for deciding whether you trust or not that particular information. 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 for enabling 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.

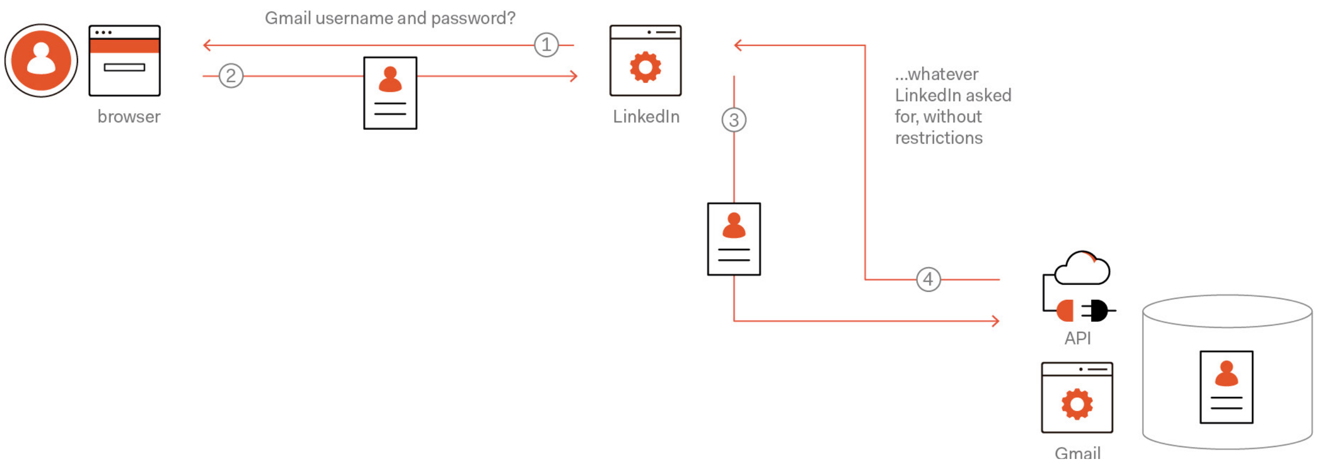
The Password Sharing Anti-Pattern

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 in LinkedIn, in whatever way they want. The mechanics of how they got signed in in LinkedIn is not the point in this scenario. Say that LinkedIn wants to suggest you to 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 course.*

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 on 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 the intent that LinkedIn had with this transaction was good (it is mutually beneficial both for me as a user and for LinkedIn as a service for me 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: OAuth2

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.

OAuth2 was designed precisely to implement the delegated access scenario described earlier, but without the bad properties we identified as part of the brute force approach. The defining feature of the OAuth2 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 on in this book.

[1] The first incarnation of OAuth was OAuth1, a protocol that resolved the delegated access scenario but had several limitations and complications. The industry quickly came up with an evolution, named OAuth2, which solved those problems and completely supplanted OAuth1 for all intents and purposes. For that reason, in this text we only discuss OAuth2.
.

Suffice 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: in the following discussion, we are assuming that the user is already signed in 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. OAuth2, as you will hear over and over again, is not a sign-in protocol.

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

In the OAuth2 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:

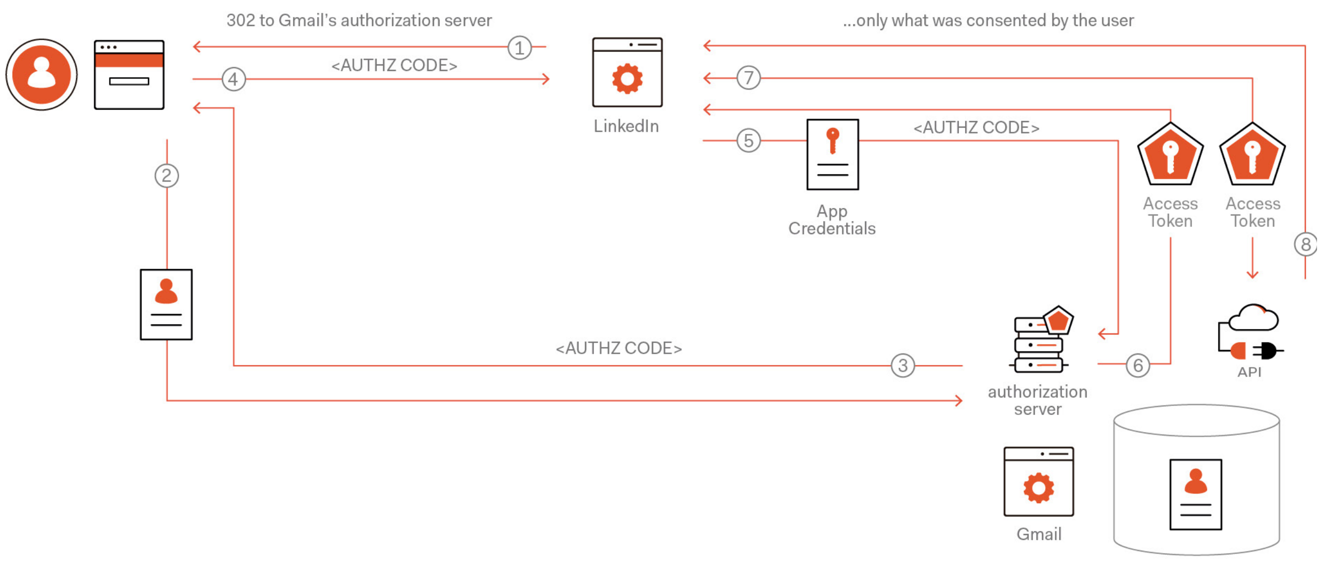


Figure 1.7

LinkedIn follows the OAuth2 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 did grant 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 for proving 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 obtained an authorization code, LinkedIn will reach out to the token endpoint of the authorization server **(5)** and will 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 that is used to grant to 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 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 described so far is the canonical OAuth2 use case, the one for which the protocol has been originally designed. In practice, however, OAuth2 is used all over the place, and it incurs in all sorts of abuses, that is, in ways in which OAuth2 wasn't designed to be used. Be on the lookout for those problematic scenarios: every time you hear that some solution uses OAuth2, please think of the canonical use case as described here first. OAuth2 supports many other scenarios, and in this book, we will discuss most of them. However, the core intent is as expressed in the use case we described in this section. Thinking about whether a solution is using OAuth2 in line with the intent expressed here, or delve from it significantly, is a useful mental tool to verify whether you are dealing with a canonical scenario or if you need to brace for non-standard approaches.

Layering Sign In on Top of OAuth2: OpenID Connect

Let me give you a demonstration of one particularly common type of OAuth2 abuse. As OAuth2 and delegated authorization scenarios started gaining traction, many application developers decided that they wanted to do more than just calling APIs. They wanted it to achieve in the consumer space, what we achieved with SAML. They wanted to allow users to sign in in 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 in LinkedIn directly, without the need to create a LinkedIn account. In other words, LinkedIn would just want users to be able to sign up in LinkedIn 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, OAuth2 was not designed to implement sign-in operations. Most providers only exposed OAuth2 as a way of supporting delegated authorization for their API, and 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 OAuth2 flows to achieve some kind of poor man's signing in. Imagine the delegated authorization scenario described for the canonical OAuth2 flow and imagine it taking place with the user not being previously signed in in LinkedIn. The following picture describes this flow:

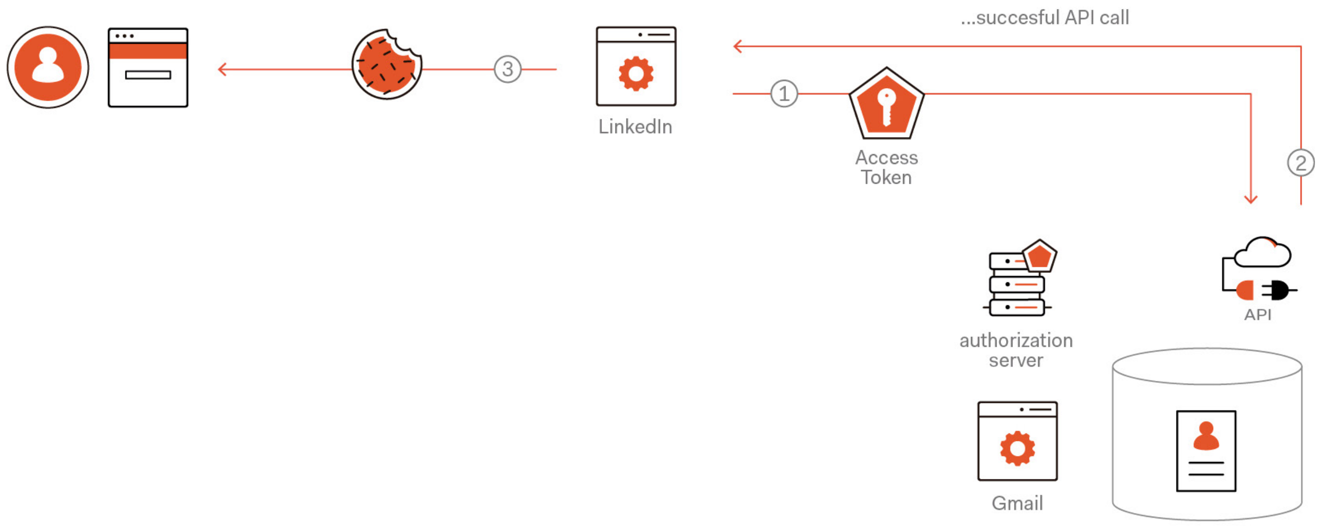


Figure 1.8

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 for successfully calling 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 in 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 the fact that access tokens are opaque to clients is that an attacker could get a token from a user and somehow inject it in 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 that I'm describing are called the Confused Deputy attack, and they are a classic shortcoming of piggybacking sign-in operations on top of OAuth2.

Even more aggravating: with this approach, there is no way to standardize the OAuth2 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 OAuth2.

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 OAuth2. I'll go into painstakingly fine details about that effort in the rest of the ebook, 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 OAuth2 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:

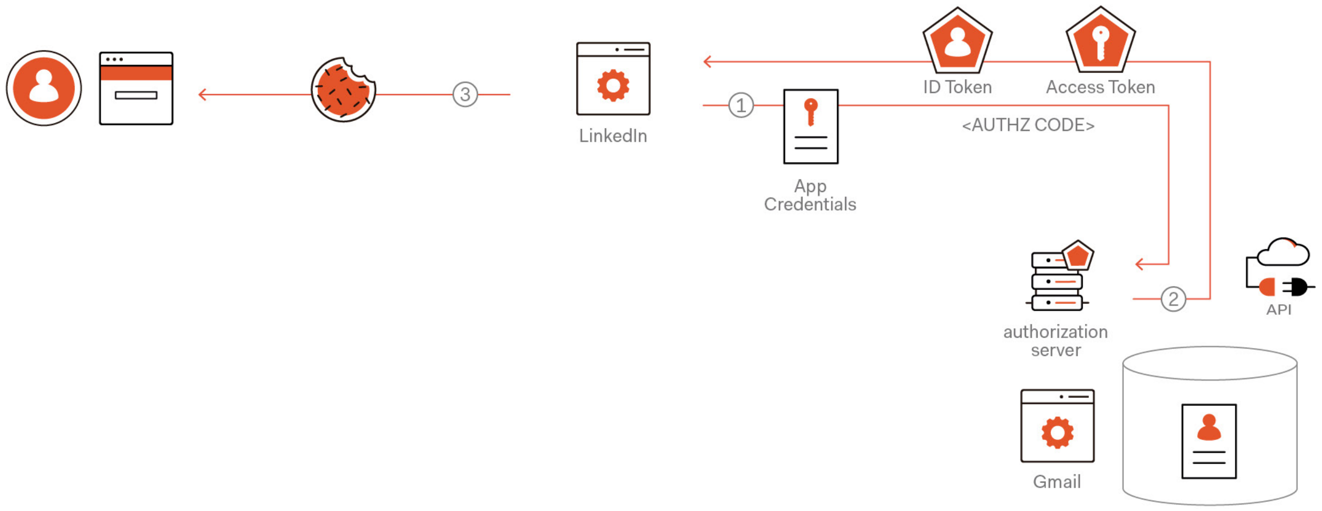


Figure 1.9

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. The use of 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 inspected and validated SAML tokens. It also means the ability to extract identity information from it, once again, just like we learned is common practice with SAML. Those properties are what makes it possible to achieve proper signing in using OAuth2. 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 OAuth2 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 in deep in 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 that you support, such as OAuth2, OpenID Connect, SAML, WS Federation.

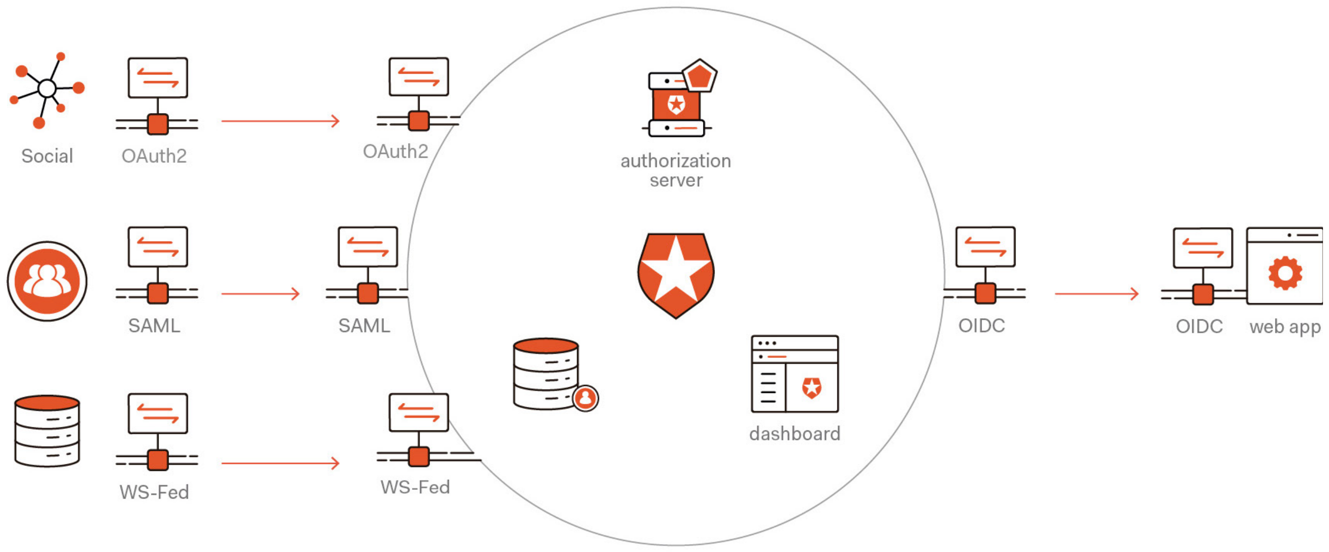


Figure 1.10

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 are using 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 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 for managing users. For example, you can have multiple accounts sourced from multiple identity providers, that accrue to the same account in Auth0 and your app. You can normalize the set of claims that 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 that if you want to execute custom logic, for example, subscription, or billing, or any functionality which just makes sense in your scenario to occur at the same time of authentication, you can easily achieve that.

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 you with a dashboard that has 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 - OAuth2 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 was 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 exhibiting 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. Ploughing 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 code OpenID Connect and OAuth2 core specs. All the specifications that you see in the constellation of OAuth, and OpenID, and JWT, and 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. There is an entire ring of best practices or new capabilities not shown here. The complete picture is, in fact, much larger.

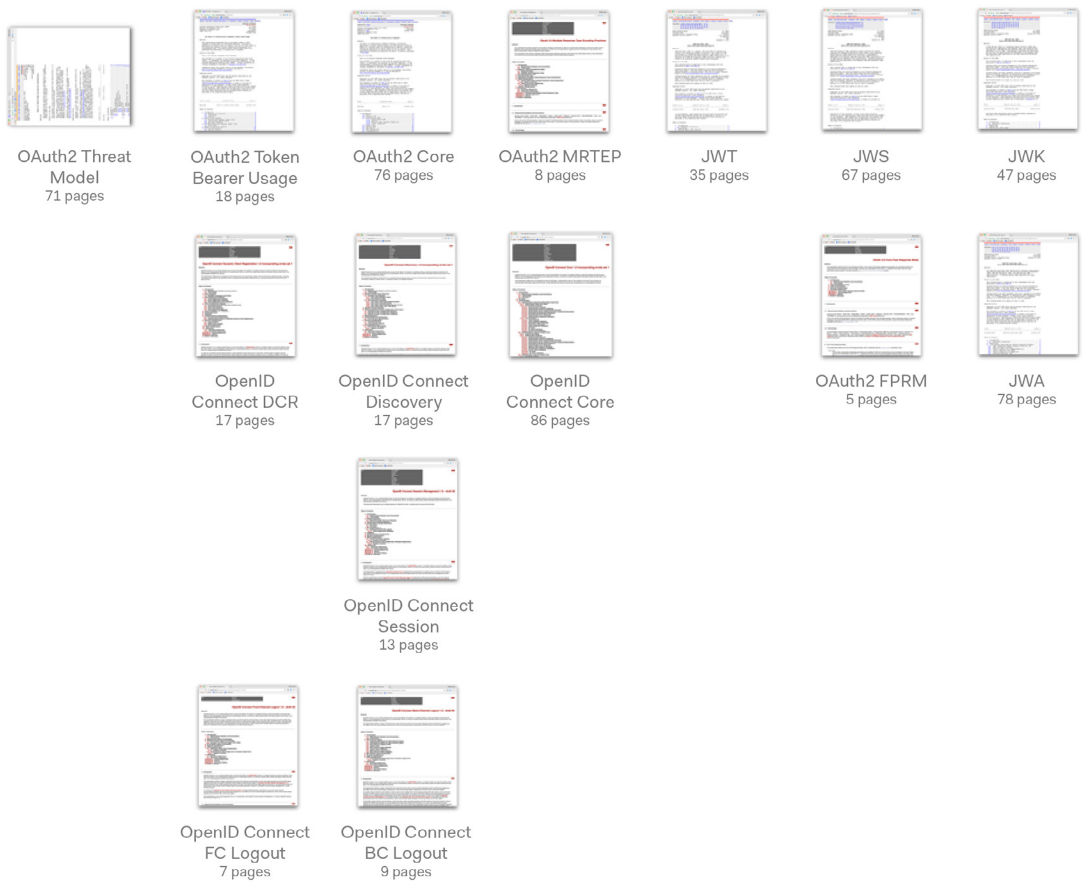


Figure 2.1

The main reason for which 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 OAuth2 and OpenID Connect specifications, and you'll be fine. Of course, the reality is quite different. If that would be true, then not a lot of 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 for getting 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 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 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. It's just that their language is such that it's not presented, usually, 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 ask 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.

OAuth2 Roles

Let's start with the few definitions that I mentioned we need before starting our scenario-based journey through the specifications. OAuth2 and OpenID Connect define a number of primitives that are required for describing what's going on during identity transactions.

In particular, OAuth2 introduces several canonical **roles** that different actors can play in the context of an identity transaction. As OpenID Connect is built on OAuth2, 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 entity that is most salient for developers. The client, from the OAuth2 perspective, is the application that needs to obtain access to the resource. In our example, that would be the LinkedIn web application.

For OAuth2, 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 OAuth2, 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 world. You can see how naming "client" the resource you want to gate access to might be confusing!
Now that you have seen in Chapter 1 how OpenID Connect was built on top of OAuth2 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 to for anything entailing 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**, that is the endpoint to which apps typically speak to in programmatic fashion, performing the operation that actually retrieves tokens.

Authorization and token endpoints are defined in OAuth2 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 that 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.

OAuth2 Grants and OIDC Flows

The most complicated things in the context of OAuth2 and OpenID Connect are usually what we call the **grants**. In a nutshell: grants are just the set of steps a client uses for obtaining some kind of credential from the authorization server, for the purpose of accessing a resource. As simple as that. OAuth2 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 to say at this point that there is a core set of grants originally defined by OAuth2: Authorization Code, Implicit, Resource Owner Credentials, Client Credentials, and Refresh Token. OpenID Connect introduces a new one, the Hybrid, which is combining two particular OAuth2 grants into one single flow.

In addition to the grants defined by the code OAuth2 and OpenID Connect specifications, the OAuth2 working group at IETF and the OpenID Foundation continuously produce independent extensions, devised to address scenarios that weren't 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, which helps the community to receive the guidance it needs to address new scenarios as they arise.

The book will examine every essential grant in details, with a particular emphasis of 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 the mechanics of it, I have to make a couple of high-level introductions of artifacts and terminology that we use in the context of OAuth2 in OpenID Connect. In particular, I want to talk to you about client types.

A **confidential clien**t in OAuth2 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 makes it possible for the app to prove to the authorization server its identity as a registered client during any request.

This typically happens with any app that is a singleton. 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 is running on a server, the server administrator is the only person that 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 for you to get a unique key that would represent that particular instance of a client.

You certainly cannot embed such 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 needs first to 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 the user of a device.

Another scenario involves long-running processes.
For example, consider a continuous integration system that uses your Jenkins and compiles your product overnight, runs tests, and similar long-running tasks. It's likely that you'll 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, it may happen that everything grinds to a halt, and no one knows why. This happens because very often people 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 the use of 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 for proving your identity as a client when you're asking for a token for accessing 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

TThe grant that 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 in a website with granting that website with 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 experience for the user 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. We see that there are security implications here and there, but, as just said, it is just like SAML.

Setting this thing up from a developer 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 that you were 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've done.

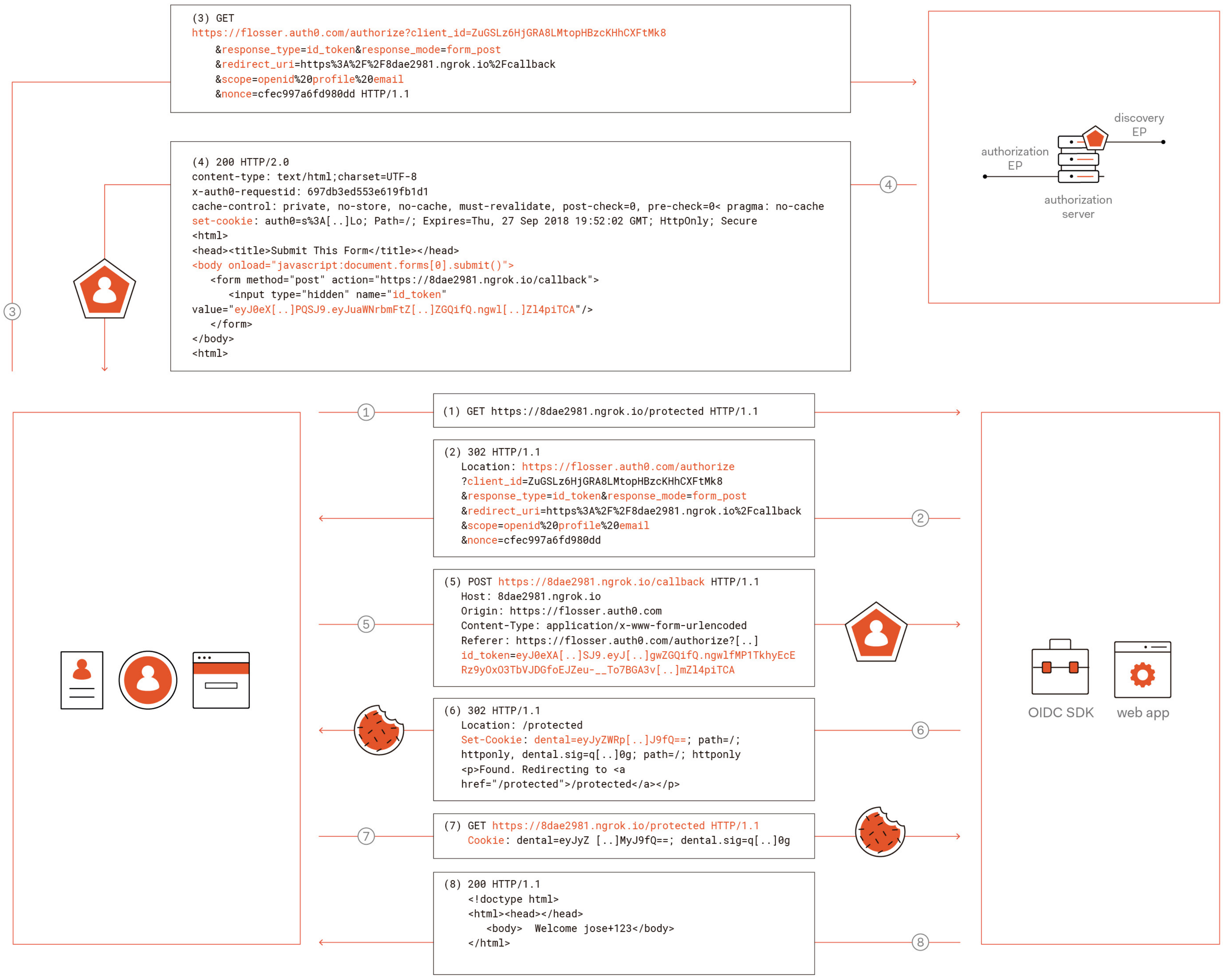
A detailed walkthrough

Let's see in detail how the implicit grant with form_post works. Take a look at the scenario shown by the following picture:

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 endpoint and the discovery endpoint. I don't show the token endpoint because, in this particular flow, we don't use it.

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



```
(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
   Rz9yOxO3TbVJDGfoEJZeu-__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

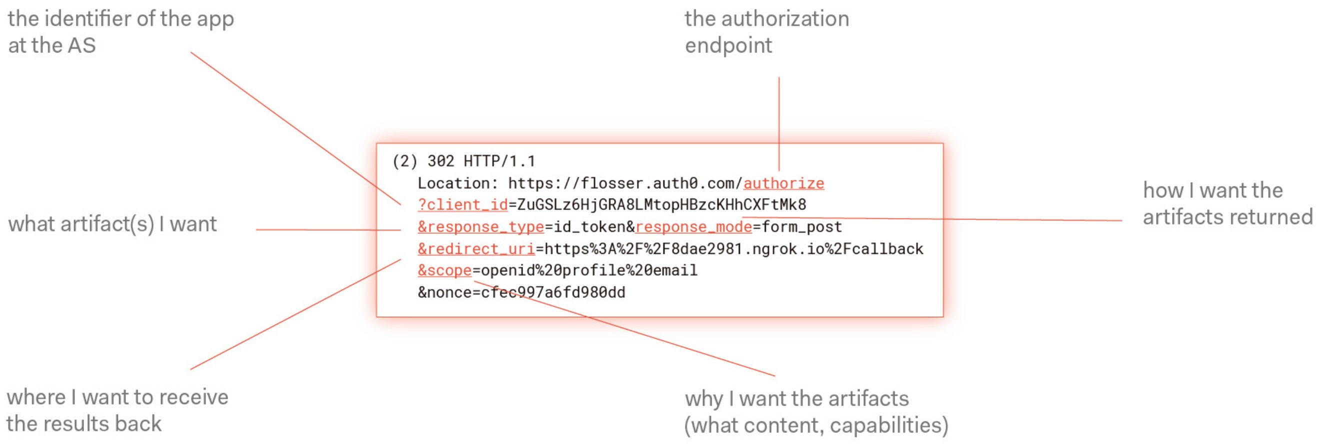
Let's see how the access plays out by describing each numbered step.

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 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, and has a number of parameters meant to communicate to the authorization server all the information necessary to perform the required authentication operation.



```
(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
```

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)

Figure 3.2

It's really important to understand the anatomy of this message since all the other messages that we'll see will be a derivative 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. There is a large variety of artifacts that I can ask for. I can also ask for combinations of artifacts: we'll see those combinations in detail.

- ✓ **Response mode**. Response mode is the way in which 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 would place stuff anywhere but in a form post, then 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 to. I need to specify this because the tokens that 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 it directly, without needing to do anything else, like 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 section about token validation (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 address they provide, 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 to malicious actors my bearer tokens, and that's all they'd need to impersonate me. OAuth2 and OpenID Connect are strict about this: the redirect URI that you specify in the request has to be an exact match of what you want.

- ✓ **Scope.** The scope parameter represents the reason for which 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 profile and email information of the user if present.

  In short, with the scope, I am specifying the reason for which I want the artifacts I am requesting. We will see that, when we will use APIs, we'll be asking 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 I 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 that I saved, and I'll be confident that the token I received is the one I requested. If I receive a token that has 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 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 OAuth2 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

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 to prompt for consent. How this occurs isn't specified by OAuth2 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 that comes back is in the format dictated by the standard. You can have multi-factor authentication, multiple pages, 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 later on you need to hit the authorization endpoint again, 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, which is what we requested. It is being returned as a parameter in the form post that we are getting. You can see in the body of 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's going to post the form to our application. This means that the requested ID token is finally 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 it likes it or not 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. Together with the cookie creation, the app emits an HTTP 302 response, which redirects 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 are requesting a protected route, but this time we present a session cookie with it.

If you compare the original request in 1 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, which is an HTTP 200 response with a page in the body.

From now on, every subsequent request toward the application will carry the session cookie, proving that there is an authenticated session in place.

Anatomy of an ID Token

As we said earlier, the ID token is an artifact proving that a 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 for which 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 OAuth2. In fact, at the OAuth2 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.

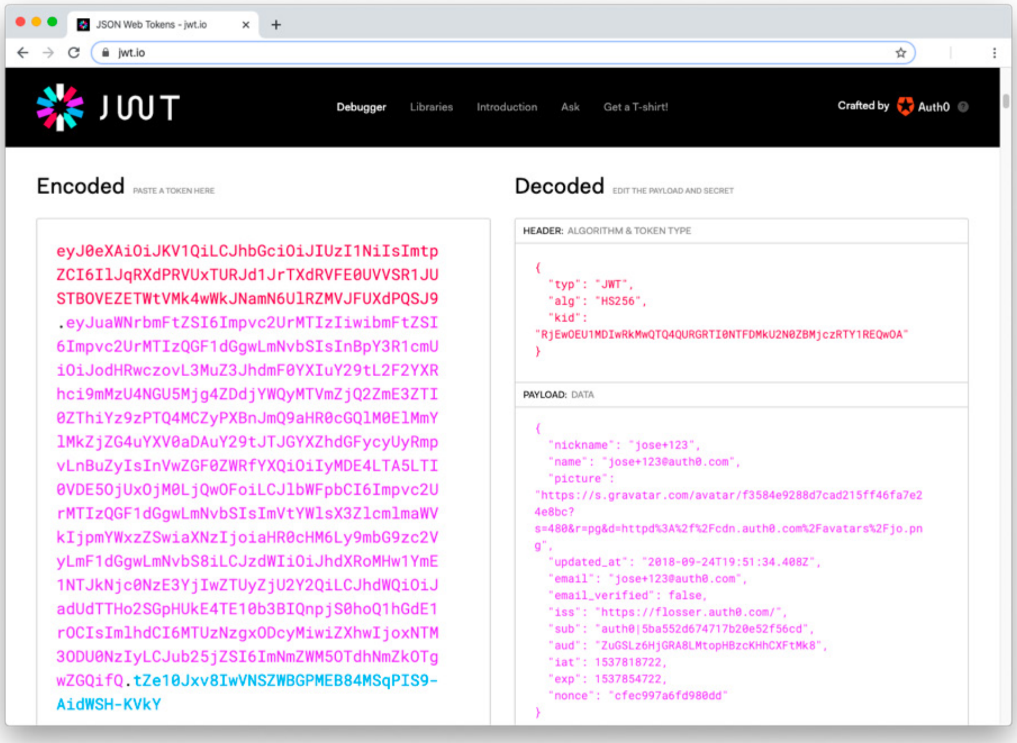
OpenID Connect defines the ID token as a fixed format, the JSON Web Token (JWT) format. The specification actually defines not just the format but the list of claims that must be present in an ID token. In addition, it even tells you in normative terms what you need to do in order to validate some of 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_LDD58XKaJJ8Z9WhPUr1RFll4IPTNEmtmgSEWXz
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:



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

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

```
{
  "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

You can see on the right side that we have 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 were expecting to retrieve. Going in more details, we have:

- ✓ The **issuer** (iss), which is a string representing the source of the token, that is the entity behind the authorization server - like the key, also found via the discovery endpoint.
- ✓ The **audience** (aud), which represents the particular application which 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 they're trying to trick the app into accepting it.
- ✓ The **issued-at** (iat) and **expiration** (exp) are coordinates that are used for evaluating whether this token is still within its validity window or if, being expired, it can no longer be accepted. We'll see during the API discussion 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 are 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 for signing in. All those scenarios entail an entity, the resource, to receive a token and make a decision about whether it entitles the caller to perform whatever operation the caller is attempting. How does the resource take that decision?

Subject Confirmation

The subject confirmation is a concept we inherit from SAML. In particular, the subject confirmation method determines the way in which a resource decides whether a token has been used correctly or not.

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 good or service you are paying for. No further questions will be asked because all it takes for using 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 that contains 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 bearer. That 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 would not be able to perform the additional signature and provide *proof of possession.*

Today nobody substantially is using proof of possession in OAuth2 or OpenID Connect. But proof of possession is now coming back. There is a specification, still in draft, which shows how to use the mechanism I just described in OAuth2 and OpenID Connect, but it is not mainstream at all. That specification is not yet an approved standard.

So, to all intent 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 OAuth2, access tokens have no format. The standard doesn't specify any format mostly because originally, it was thought for a scenario where the authorization server and the resource server are co-located, and they 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 happens to be the primary key in a specific table where the authorization server 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 for finding in the database the correct row and in it the consented permissions. The resource server uses that information for making an authorization decision.

This scenario is compliant 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 happens typically when the resource server and the authorization server are not co-located, 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 which apply pretty much to every format, and in particular, to JWT:

- ✓ **Signature for integrity.** Your token is signed, and we have seen the reasons for which we want to sign a token: being sure of the token origin and preventing tampering in transit. The token must provide some indication about the key and the algorithm used in order for its recipient to be able to check its signature.
- ✓ **Infrastructural claims.** Token formats will typically include infrastructural claims, meant to provide information that 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 *audience* 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 it 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 can be exposed by authorization servers. 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, send 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 that 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, normally, it's just better if you validate your own token at your API.

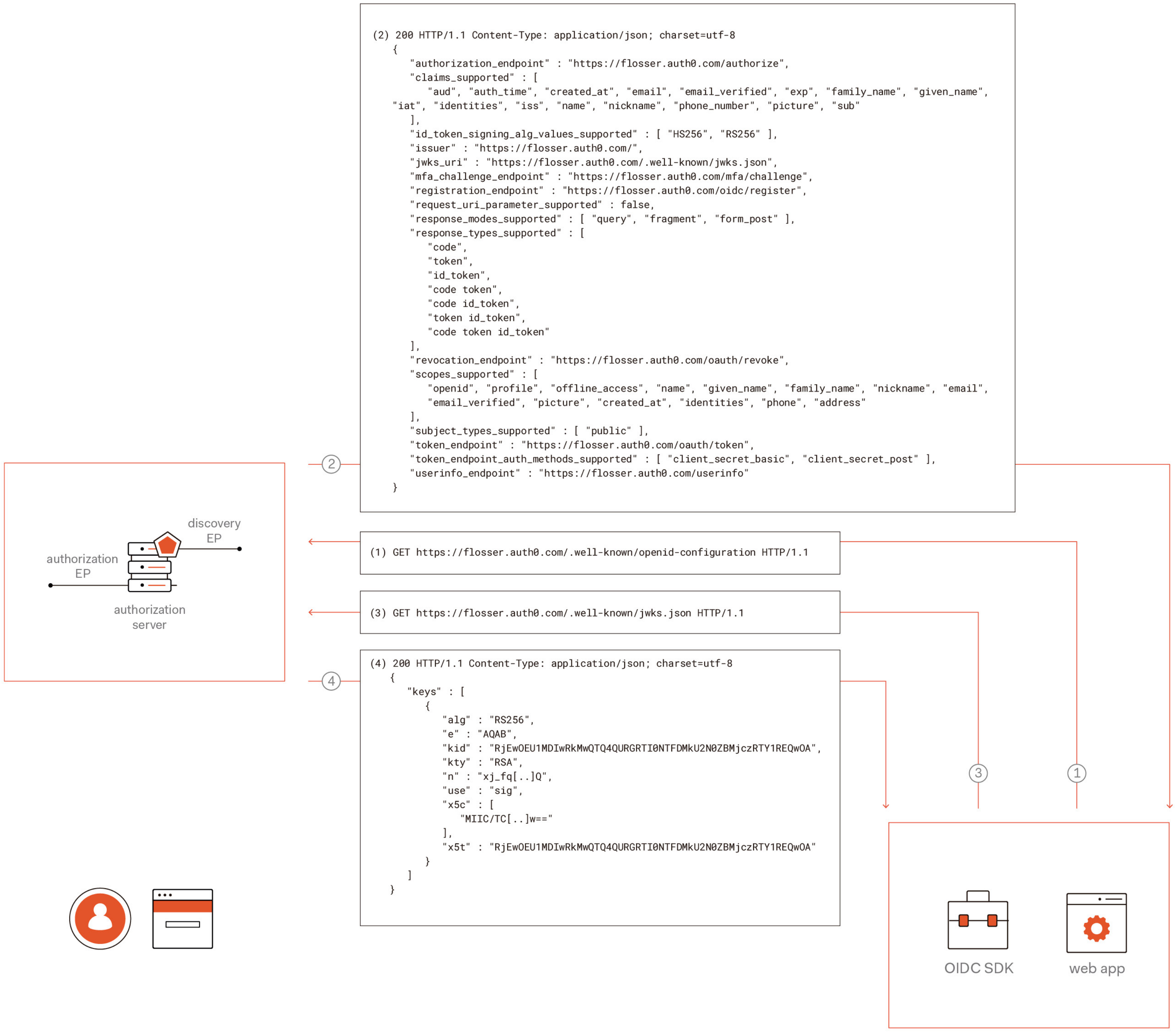
.

Metadata and Discovery

The way in which token validation middleware discovers the values expected in valid tokens is 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 that contains 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.



```
(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"
    }
```

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

```
(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" : "RjEwOEU1MDIwRkMwQTQ4QURGRTI0NTFDMkU2N0ZBMjczRTY1REQwOA",
            "kty" : "RSA",
            "n" : "xj_fq[..]Q",
            "use" : "sig",
            "x5c" : [
               "MIIC/TC[..]w=="
            ],
            "x5t" : "RjEwOEU1MDIwRkMwQTQ4QURGRTI0NTFDMkU2N0ZBMjczRTY1REQwOA"
         }
      ]
    }
```

Figure 3.5

1. Request Configuration

At load time or even the first time that you receive a message, the middleware reaches out to 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 claims which are supported (*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 would be to actually make a GET request to the address at which 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

COMING SOON

Chapter 5 - Desktop and Mobile Apps

COMING SOON

Chapter 6 - Single Page Applications

COMING SOON