THE OPENID CONNECT HANDBOOK

By Bruno Krebs

Auth0

Index

Introduction

In this book, you will learn about OpenID Connect, a protocol that helps applications of all types handle end-user authentication and verify the identities of these users. Before jumping into the protocol, you will be introduced to topics like authentication, authorization, entity, and identity. These topics will set the stage as you learn about OpenID Connect and related technologies.

After this introduction, you will dive deep into why OpenID Connect exists, what sets it apart from similar protocols, and the pieces that compose this technology. Later, you will learn how OpenID Connect is applied in practice, with examples of how to authenticate and check the identity of users in different application architectures, such as regular web applications, Single-Page Applications (SPAs), and native apps.

By the end of the book, you will have learned what OpenID Connect is, how and when you can take advantage of this protocol, and how it works. This knowledge will give you enough foundation to use the protocol in an informed way, to extend it to your needs, and to troubleshoot problems that you might face.

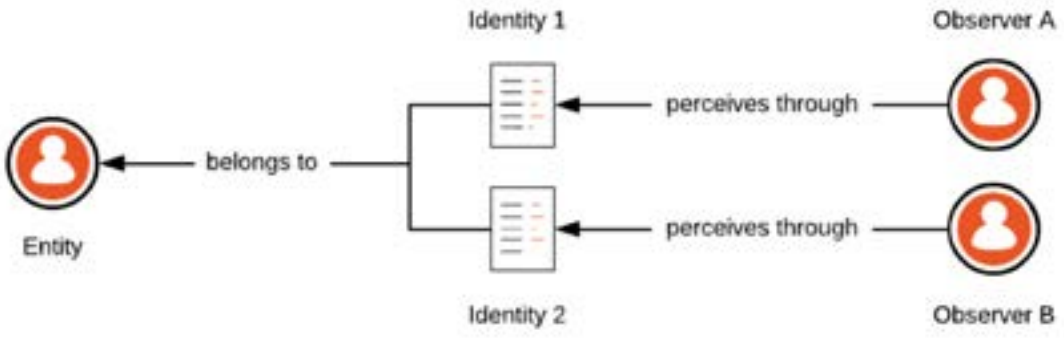
A Word on Entity and Identity

Although the terms entity and identity frequently appear in our daily lives, people often misunderstand and misuse them. For the sake of this book, what you need to know about them is that **entity** refers to a *thing* that exists as an individual unit while **identity** is a set of attributes that you can use to distinguish this entity within a context. If this definition didn't help you clarify these terms, a couple of examples may do a better job.

For starters, take into consideration any person — Einstein, for example. Einstein, by the most fundamental definition, was an entity. In this case, he was an entity that was capable of thinking, speaking, walking, teaching, and so on. However, the people around the scientist didn't perceive him as an entity directly. Those people perceived Einstein indirectly through his various attributes, like his name, gender, height, and home address.

The different subsets of attributes that people used to perceive Einstein formed the identities that he had. For some people, he was a professor with a name and an educational background (among other things). For a bank, he was a customer with a signature and an account number.

Another example that can help you understand what entity and identity represent and how they are related is by analyzing a software application. Just like Einstein, an application is an entity that exists on its own (i.e., independently of the perception of others). Also like Einstein, being an entity, the application has multiple identities. For a SPA consuming this application, its identity would consist of an internet domain, a TLS certificate, and so on. For a database, the identity of this application would be a set of credentials (like username and password) and the permissions it has (for example, what tables the application can access).



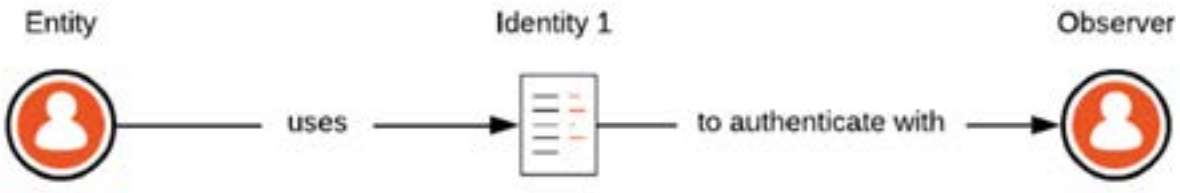
The image above illustrates two different observers perceiving some entity through two different identities. That is, each observer uses a different set of attributes (identities) to understand the same entity.

The most important concept to take from this section is that these identities will be formed by the attributes that are relevant to the context in which the entity is inserted. This definition will be important when you learn how you can use OpenID Connect to identify users.

Authentication vs. Authorization

Two other topics to cover before diving into OpenID Connect are authentication and authorization. Like the terms in the previous section, these topics can cause great confusion and are often misused. As such, it is important for you to understand the meaning of these terms and how they interconnect.

For starters, authentication is the process of confirming the identity of an entity (e.g., a user). An authentication process usually relies on some form of proof. For example, if you go to the bank and try to withdraw money from your account, the clerk might ask you for an ID (an official document) to check who you are. Along the same lines, if you buy a flight ticket, you might need to use a passport to prove you are the person entitled to that ticket before hopping on the plane. Both examples illustrate real-life situations where authentication processes take place to confirm your identity.



In contrast, authorization refers to the process of verifying what entities can access, or what actions they can perform. For a concrete example, imagine a situation where you buy a ticket for a show. In this case, more often than not, the establishment will not be interested in your identity (i.e., on who you are). What they care about is whether you are authorized or not to attend the show. To prove that you have the right to be there, instead of using an ID or a passport, you would use a ticket that contains no information about you.

Although the explanations above can shed some light on the topic, the definition and usage of these terms might frequently overlap. For example, if you give the bank scenario more thought, you will realize that, in this case, the identity of the person is also used to authorize the account owner access to the money. That is, the clerk wouldn't authorize any other person to get your money, even if they were carrying your ID and if they knew how to replicate your signature.

What is important for you to understand here is that authentication can lead to authorization but that the opposite is not true. Although proof of identity might be enough for you to get access to something (i.e., to be authorized to achieve something), having authorization cannot be used to identify an entity (like in the example where you would buy a ticket for a show). In this scenario, having a ticket in hand

doesn't confirm your identity (e.g., your name or age). All the ticket proves is that you have the right to join the show, nothing else.

Recap

In the first chapter of this book, you were introduced to topics like entity, identity, authentication, and authorization, which provide a foundation for understanding OpenID Connect. In summary, you learned that any entity can have multiple identities (e.g., Einstein can be perceived as a professor or as a customer), and that identities are sets of properties that belong to the entity in question. Besides that, you read about authentication and authorization and how they relate. More specifically, you learned how an entity can use its identity to gain authorization to perform some action, but that the other way around is not possible (i.e., having authorization doesn't mean being authenticated or identified). With this knowledge fresh in your mind, you are ready to start learning about the OpenID Connect protocol and how to use this technology to deal with end-user authentication.

OpenID Connect Introduction

OpenID Connect, popularly abbreviated as OIDC, is a protocol that enables different types of applications to support authentication and identity management in a secure, centralized, and standardized way. Apps based on the OpenID Connect protocol rely on identity providers to handle authentication processes for them securely and to verify the identities (i.e., personal attributes) of their users.

- **Key Term:** *If you read the spec, you will see that OpenID Connect doesn't employ identity provider in it, even though this term is widely used by the digital identity community. Instead, the protocol uses Authorization Server to refer to the entity in charge of authenticating end-users. This choice may seem awkward, but there is a perfectly good reason for it which you will learn about soon.*

Imagine, for example, you have a restaurant application that allows authenticated users to book tables. By using OpenID Connect, instead of dealing with the credentials of these users, your app would offload the authentication process to an identity provider (for example, Google, Microsoft, or Auth0). More specifically, when visitors start this process, your application would redirect them to the identity provider of choice where they would authenticate themselves to prove their identities. After the authentication process, your app would get this identity proof and would allow users to book their tables based on it.

How these identity providers handle the authentication process is out of the scope of OpenID Connect. That is, from the perspective of the protocol, identity providers are free to decide if they handle user authentication through a set of credentials (e.g., username and password), if they enhance the security of the process by using features like multi-factor authentication, or even if they relay this process to other identity providers and other protocols. What OIDC defines is how identity providers and applications interact to establish end-user authentication in a secure way.

Authentication Protocols in a Nutshell

After reading the introductory section on OpenID Connect, you might have wondered, "why would one care about a protocol?" Or "why not just keep handling usernames and passwords locally?" Or, even more, "there are other protocols that solve this very same problem, why not use them?" To answer these questions, you will need to learn a bit about how the digital identity scenario evolved throughout the years.

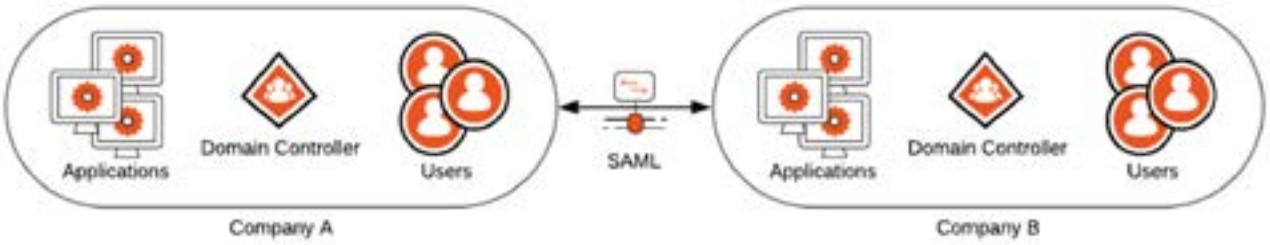
In the beginning, there was chaos (I mean, passwords). Before humanity invented computers, people used passwords to control who had access to certain areas. When we created the first computers, we soon noticed that we would need a way to secure these machines, and we thought "why not make these machines support passwords?" In the early days of this new era, using passwords worked just fine. Anyone that wanted to use a machine or an application would have to know the password, and they would have to be there, physically.

But then, we created computer networks, and "being there" was no longer a requirement. When this happened, organizations around the world soon realized that letting computers and applications handle authentication on their own was not such a good idea, for multiple reasons (for instance, users would end up being duplicated all over the place). That's when the first authentication protocols, like Kerberos, appeared.

At that time, computer scientists decided to centralize authentication on domain controllers (servers that respond to security authentication requests like login, permission checking, etc.). The biggest problem with these domain controllers (as defined by the first protocols) was that they were conceived to put all the entities (end-users, applications, and services) involved in the process under the control of a single organization. That is, these protocols were not created to handle scenarios where users want to connect to third-party clients as in a typical cloud computing scenario, where the organization that handles the identity provider typically does not have knowledge or control over those clients.

It was not long before the IT community realized that this strategy was incomplete and that a better solution was needed. At that point, big IT players assembled a committee to solve the problem. The solution they come up with was SAML (Security Assertion Markup Language), a standard for exchanging authentication and authorization data between different security domains. With SAML, a user that belonged to company A could consume a service on company B in a reliable way and without requiring companies to duplicate user profile. The protocol was conceived in a way that enabled the service provider (company B in this case) to validate the user identity on the fly.

* **Note:** *Both SAML and OpenID Connect take advantage of digitally signed tokens to carry end-user personal attributes around. These digital signatures enable third-party applications to confirm the veracity of the information on the fly (i.e., without having to issue yet another request to the identity provider to check the data).*



SAML, on its own, solved most of the problems OpenID Connect solves now. However, one big problem inherent to this protocol is that it is heavily based on XML. The issue with XML is that its flexibility becomes a problem when you need to digitally sign identities, where two elements listed in a different order can break a signature verification. The XML format is also extremely verbose. This makes it too heavy and too powerful for most scenarios, where applications just want to authenticate end-users and get a few bits about their identities, ideally even when a high-speed connection is not available or when the devices involved are not that robust.

So even though, technically speaking, SAML provides enough to cover similar scenarios to the ones OpenID Connect addresses, the protocol ended up being used mostly in the business world. In the consumer world (social networks, for example), there was another effort called OAuth (Open Authorization).

You will soon read more about OAuth and why this technology is important but, in short, this protocol was created specifically to handle **delegated authorization** scenarios, like when you let a random application post something on Facebook as if it was you. However, it got so much traction that developers started using it to do things it was not created to do, like handling end-user authentication. This led to an effort that resulted in OpenID Connect, a protocol that extends OAuth 2.0 to address authentication.

The above story, although very summarized, contains enough information to answer the questions that started this section:

- **Why would one care about an authentication protocol?** Mainly because developers want to solve identity management in a secure and interoperable way.
- **Why not just keep handling usernames and passwords locally?** Because most of the time, people need to reuse existing accounts to access applications and because these apps need to perform actions for users in other places.
- **There are other protocols that solve this very same problem, why not use them?** Because they might not technically support scenarios where OpenID Connect is used (like cloud environments) or because they might be too "expensive" to solve the problem.

How OpenID Connect Works

There are two different perspectives you can use to analyze how the OpenID Connect protocol works: the end-user perspective and the software perspective. From the end-user point of view, the steps involved in an OIDC flow are fairly simple. For starters, users will open the application in question to start the authentication process. When they start this process, the application will redirect them to an identity provider where they will authenticate through whatever means the service requests. After they authenticate, the identity provider will redirect them back to the application where they will be logged in.

The description above is an oversimplification of the process, but it is sufficient to illustrate what end users see. However, as you will learn, developing applications that rely on OpenID Connect to authenticate users involves more steps and moving parts, and these steps will vary depending on what platform your application will run.

For example, consider a traditional web application (those that do a full reload on the browser for each page requested). From the perspective of this type of application, the steps involved in authenticating users are:

1. Visitors request the app to start the authentication process
2. The application redirects visitors to the identity provider
3. The identity provider redirects visitors back to the application with a few artifacts
4. The application uses these artifacts to issue a request to the identity provider to complete the authentication process
5. The application shows a page to users containing some indication that they are logged in

* **Key Term:** *One important artifact that your application will get at the end of an authentication process is an ID Token. This token will contain a set of personal attributes about a user and, as such, your application can use it to identify the user (hence the name, ID Token).*

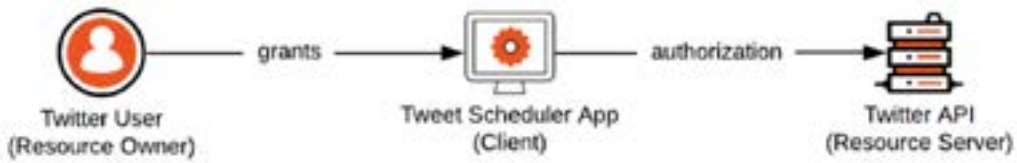
When you dive deeper into the protocol, you will see that the steps above are also oversimplified. However, in broad terms, they show how authentication processes flow and what steps you, as an application developer, can expect to see.

One thing that was briefly mentioned is that OpenID Connect is based on a framework called OAuth 2.0, which handles delegated authorization. If you know how this framework works, you probably realized that the steps above are extremely similar to what happens in an OAuth 2.0 authorization flow. The similarities between the processes are not a coincidence; they are a reflection of that fact that one extends the other. In the next section, you will take a quick look at OAuth 2.0 and how this framework handles authorization.

OAuth 2.0 Overview

Using the official terminology, OAuth 2.0 is an authorization framework that enables **clients** to use **resource** servers on behalf of **resources owners**. Clients, resource servers, and resource owners are probably not terms you are used to, so this might not help explain what the framework does.

In that case, let's take into consideration the following example. Imagine you are using a third-party application that will schedule tweets for you so you don't have to "be there" to press the "Tweet" button. In this scenario, the application that posts the tweets would be what OAuth 2.0 calls a Client, you would be the Resource Owner (the owner of the tweets), and the Twitter API would be the Resource Server (the place where your tweets reside).



Now, with this scenario in mind, if you replace the terms that the framework uses with the terms you know, the definition would read like: OAuth 2.0 is an authorization framework that enables applications to use APIs on behalf of users. This definition, while less technically precise, is probably a bit easier to understand than the one that started this section.

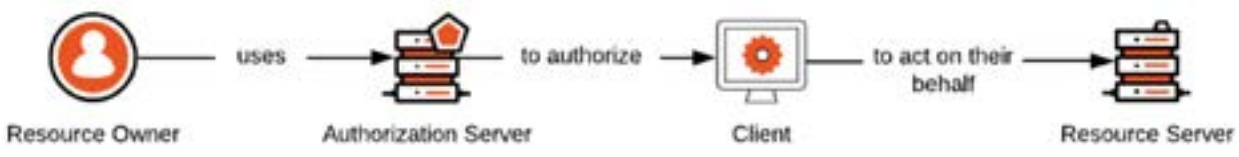
Apart from the elements mentioned above, another essential term that the OAuth 2.0 framework defines is **Authorization Server.** At the very beginning of this chapter, we said that OpenID Connect uses authorization server to refer to the entity in charge of authenticating users. We were able to say that because OIDC reuses (or extends) the term authorization server to include its role as the place where end users sign in. However, in a pure OAuth 2.0 sense, this entity plays a slightly different role. Authorizations

servers in an OAuth 2.0 flow provide a means for resource owners to decide whether they want to grant the client the power to do something on their behalf or not.

Revisiting the scenario where you would grant an application permission to tweet on your behalf, the authorization server would be Twitter itself. In this case, the first time you asked the third-party application to schedule some tweets, the app would send you to Twitter where the service would ask you if would like to give the app this consent. If you were to agree, Twitter would provide the application with an artifact representing this grant. Then, when the time to post one of your scheduled tweets arrived, the third-party app would send a request to the Twitter API and would present this artifact to let the resource server know that you gave the app permission to perform this action.

* **Key Term:** *The OAuth 2.0 framework uses the term **Access Token** to define the artifact that grants third-party applications (Clients) delegated authorization to act on behalf of users (Resource Owners). The name is quite descriptive. With this token, an API (Resource Server) will let an application access some part of it to perform some action.*

The first part of the scenario described above (the one where you tell Twitter that you allow the third-party application to tweet on your behalf) resembles what you learned so far about authenticating end-users with OpenID Connect. In this step, you are redirected to the authorization server, you perform some action there, and then the application gets an artifact related to you. The biggest difference is that, in a pure OAuth 2.0 scenario, the result will be an artifact that grants delegated authorization instead of an artifact that contains personal attributes about you.



This section summarizes what OAuth 2.0 is capable of doing and how it works. To learn all the details of how this framework functions would take a whole other book, but nevertheless, the key takeaways are:

- OAuth 2.0 is an authorization framework that handles delegated scenarios. That is, it does not cover all kinds of authorization scenarios.
- Since the only thing it handles is authorization, the framework does not support end-user authentication by itself (as the previous chapter illustrated with the scenario where you would buy a ticket for a show).

OpenID Connect Use Cases

Now that you are starting to get more acquainted with OpenID Connect, it is time to learn in what scenarios you can take advantage of this protocol. Basically speaking, there are three situations where using OIDC would make a lot of sense.

First, you can use OpenID Connect to enable your users to reuse their accounts on an identity provider. That is, instead of asking users to create yet another account – which would mean asking them to remember another set of credentials (or, even worse, reusing usernames and passwords) – you could take advantage of OIDC to integrate with an identity provider like Google or Microsoft to let them reuse existing accounts.

There are multiple advantages for you and your users in this scenario. For starters, as mentioned, your users wouldn't have to create another account. This would also mean that the signup process would be smoother, resulting in fewer dropouts. Related to that, using OIDC like this would make it easier for your app to ask for personal information about users. As you will learn soon, OpenID Connect defines a set of profile data categories that you can use to acquire more information about your users.

Another scenario where OpenID Connect can be useful is when the protocol is used to create a hub of identity providers. In this scenario, instead of making your application communicate with multiple providers, you can make it connect to a single one that works as a hub for the others. For example, if you check Auth0's documentation, you will see that they support this scenario and that the company integrates with more than fifty identity providers. As you can imagine, it is much easier to support a single identity provider that acts as a hub than to support each one of them separately.

The third scenario where OpenID Connect can be really helpful is where it works as a proxy for other protocols. For example, you can make an OpenID Connect identity provider work as a proxy for a more restrictive protocol like SAML. The beauty here is that, by using this approach, you can make a resource-constrained device integrate with a SAML identity provider through OpenID Connect.

As you can see, OpenID Connect opens a range of possibilities that can help you make identity management easier and more extensible. This helps you dealing with user directories by making the process easier and centralized for your IT department and by demanding less from the users that want to take advantage of your application.

Recap

This chapter covered a lot of information, so a small recap of the main points might help you moving forward. For starters, you learned that OpenID Connect lets applications offload the authentication process burden to identity providers. After that, you took a quick look into how authentication protocols evolved and why OpenID Connect was created. Then, you learned a few more details about how the protocol handles end-user authentication. In the end, you learned about OAuth 2.0 and how it relates to OpenID Connect.

These sections also introduced some key terms that are defined both by OpenID Connect and by other protocols like SAML and OAuth 2.0. Among them, the most important are:

- **Authorization Server** – the place where, in a pure OAuth 2.0 authorization flow, end-users authorize third-party applications to act on their behalf; or, in an OpenID Connect flow, where end-users authenticate.
- **ID Token** – an artifact that carries personal information about end-users that authenticate on an OpenID Connect flow.
- **Digital Signature** – the mechanism that allows third-party applications to confirm the veracity of the information they find in ID Tokens.
- **Delegated Authorization** – a particular type of authorization where an entity enables another entity to act on its behalf.
- **Access Tokens** – the artifact that grants, to a third-party application, the authorization to act on behalf of a resource owner (e.g., an end-user).

With this knowledge, you are ready to move on and get your hands dirty.

OpenID Connect in Action

In this chapter, you will learn how to use OpenID Connect in different types of applications. The goal here is to teach you how the authentication process works from start to finish and what it takes to log in a user when leveraging this protocol. With this knowledge, you will be able to understand OpenID Connect more thoroughly and will be able to debug things when they don't work as expected.

Another goal here is to show you how SDKs can help you get things done more easily. While using an identity provider like Auth0 or Google, you can offload most of the steps shown here to the SDKs they support. In fact, instead of coding and handling all the details of the protocol, you are encouraged to use these SDKs. Doing so will not only make your life easier while developing but, as identity experts constantly check these SDKs, will make your applications more secure as well.

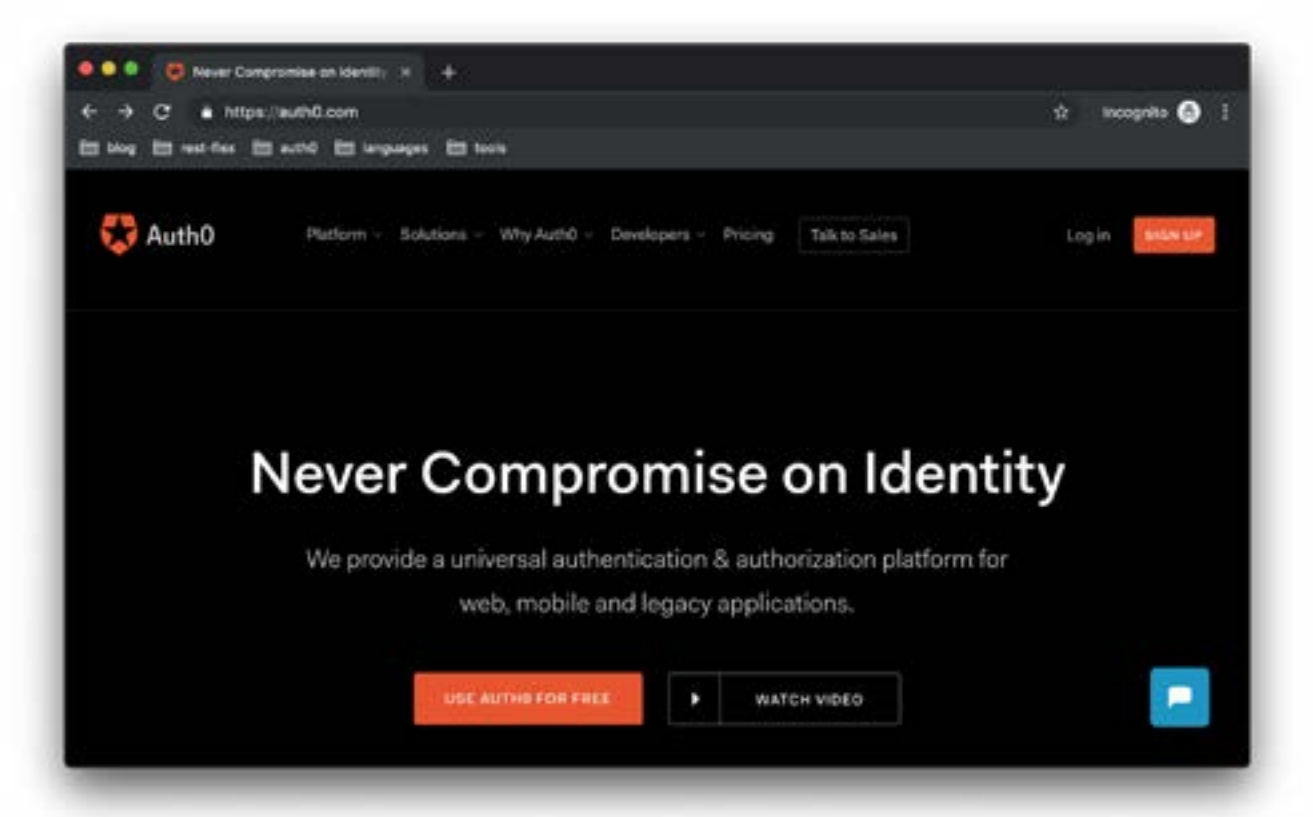
With that in mind, this is how the chapter will flow. For starters, you will create a free Auth0 account so you can use it as your OpenID Connect provider. After that, you will install some dependencies you will need in the different sections of this chapter. Then, you will learn how to handle authentication with OpenID Connect on regular web applications (single-page apps and native apps will be covered in the next version of this book, which we will publish soon). Each application type will have its own section so you can use them for future reference.

* **Note:** *Although OpenID Connect is a standard that multiple identity providers adhere to, there might be some differences in their implementations that can get in your way if you try to use the code shown here with a provider other than Auth0. However, you are more than welcome to try and let us know what the outcome was.*

Creating an Auth0 Account

To be able to focus on how different types of applications deal with OpenID Connect, you will need an identity provider that supports the protocol. There are, at least, a handful of providers you could use for this task (e.g., Google and Microsoft). Among them, a popular choice is Auth0, an identity as a service provider that features a rich community, thorough documentation, and a generous free tier.

If you don't have an Auth0 account yet, head to the company's website and click on the Sign Up button.

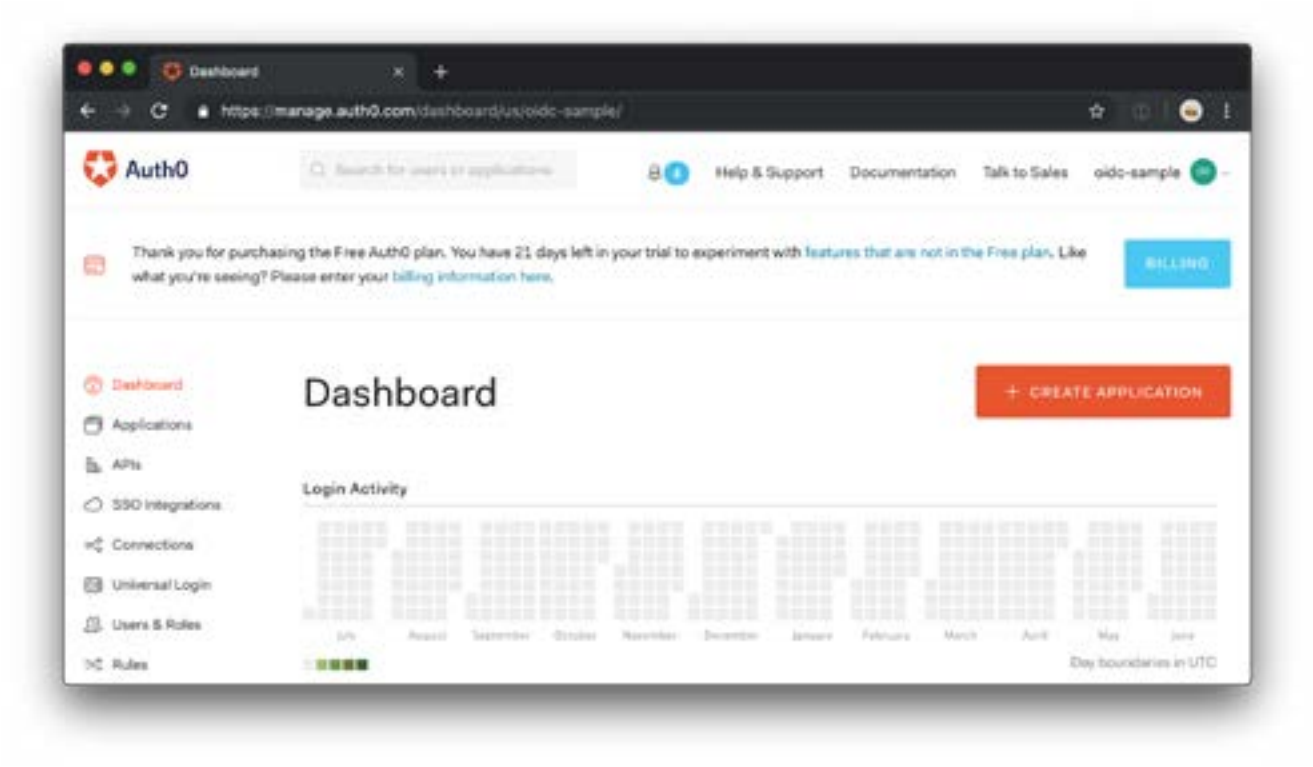


When you do so, choose one of the options available to create your account (you can reuse an existing Google or Microsoft profile, for example); then, on the next screen, choose a domain for your identity provider (e.g., oidc-sample.auth0.com) and a region for your tenant (at the time of writing, the options

are Europe, Australia, and the United States). After choosing these options, click on the Next button to continue to the next step.

* **Note:** *Not sure what region to choose? A good approach would be choosing a region that is situated near you and your servers, geographically. This strategy will make the latency between your application and the identity provider lower, increasing the speed of the overall process.*

On the next page, Auth0 will ask you to define your account type. Feel free to choose the option you prefer, fill in the information needed, and click on Create Account. After clicking on this button, Auth0 will take you to the dashboard of your account. Getting there means that you have an OpenID Connect provider that you can use on the next sections.



Prerequisites

The sections that follow will dive deep into code so you can learn the inner workings of OpenID Connect. As such, to make these sections flow as smooth as possible, you will use a set of technologies that are arguably the most popular available nowadays: Node.js and NPM. That is, you will use JavaScript to implement OpenID Connect in different types of applications. Also, to avoid making you spending time on mundane tasks like scaffolding new apps, you will use GitHub to fetch some pre-existing code.

Node.js and NPM are essential to the tasks that follow. Therefore, you will have to install them in your environment if you want to get your hands dirty. If you are not sure whether these tools are available or not, you can open a terminal and issue the following commands:

```
node -v
npm -v
```

In case you get a message back saying "command not found" or similar, head to the Node.js and NPM documentation and follow the steps to install these tools. Besides that, to fetch the pre-existing code from GitHub, you will have two options: either use the Git command-line interface as the sections will show, or use GitHub's website. Both alternatives are valid and will get the job done. However, you will probably find that using Git (which requires it to be available in your environment) is easier and more straightforward than using a webpage.

OpenID Connect and Traditional Web Applications

In this section, you will learn how to use OpenID Connect to handle end-user authentication in traditional web applications. The section will start by teaching you how to integrate an application with an OpenID Connect provider the hard way; then, it will show how you can make things easier by leveraging official SDKs.

The web app you will use in this section contains a few endpoints that will allow users to authenticate through your OpenID Connect provider and to see the result of the authentication process. That is, after authenticating, users will have access to the ID Token the provider generated for them and to the contents of this token.

To avoid spending time on mundane tasks like scaffolding a new web application, you will fetch one that has the foundation needed for this chapter (like dependencies and basic files). Then, on top of this foundation, you will add all the code needed to execute an authentication process from start to finish.

In summary, this is what you will do to handle this scenario:

1. Download a GitHub repository with basic files and the list of dependencies
2. Install the dependencies locally
3. Fetch information about the OIDC Provider from the Discovery Endpoint
4. Implement the /login endpoint that will trigger the authentication process
5. Handle the authentication /callback (i.e., retrieve ID Tokens)

6. Validate ID Tokens and let users see their contents through a /profile endpoint

OpenID Connect and the Authentication Flows

Before you move on, one thing to know first is that the OpenID Connect protocol defines different strategies to authenticate users. In this particular section, you will learn about two of them: the Implicit Flow and the Authorization Code flow. The main difference between these flows is how applications get ID Tokens and Access Tokens.

In the implicit flow, tokens are handed to the applications directly by the authorization endpoint (i.e., when the authorization server redirects users back to the application). In the authorization code flow, applications first get authorization codes (hence, the name); then, they have to exchange these codes for the tokens they need.

While learning the protocol the hard way (i.e., without SDKs), you will use the implicit flow. Using this flow will make the process a bit easier because you will be able to skip one extra step required by the other flow: the step where the application exchanges authorization codes for tokens.

While learning about the official SDK supported by Auth0, you will use the authorization code flow. Even though this flow requires an extra step, you will see that the effort needed to integrate apps with OIDC providers is minimal when compared to doing things manually.

Bootstrapping the Project

The repository that you will download from GitHub uses Node.js and Express to define a web server. As you already installed Node.js and NPM in your machine, you can go ahead and fetch the repository. If you have Git installed locally, open a terminal where you would like to save your project and issue the following commands:

```
git clone https://github.com/auth0-blog/oidc-book-regular-webapp.
git
cd oidc-book-regular-webapp
```

If you don't have Git installed, head to this URL and use the green button to download the project. After downloading it, unzip the project, and open a terminal pointing it to the project root path.

Now, inside the project root, use NPM to install the dependencies:

```
npm install
```

To make sure everything is in place, execute npm start and open http://localhost:3000 in a browser. If things are working as expected, you will see a screen that describes what this project is about and a broken link that you will use to let end-users trigger the authentication process.

Register a Regular Web App with Auth0

To enable your web application to handle end-user sign in with OpenID Connect, you must first register it with Auth0. This section will show you how to register a regular web application using Auth0's Dashboard.

For starters, navigate to your dashboard and click on the Applications section. When you get there, click on the Create Application button, and fill the form as follows:

- **Name**: OIDC Regular Web App
- **Application Type:** Regular Web Applications

Now, hit the Create button and, once Auth0 finishes creating the new application, head to its Settings section. There, the only change you will have to make is to set http://localhost:3000/callback on the Allowed Callback URLs field. This configuration is important from a security point of view because it restricts what URLs the OpenID Connect provider is allowed to call after a successful authentication process.

After changing this configuration, scroll to the bottom of the page and hit the Save Changes button. You can leave the page open since you will need to copy some information from it later.

Fetching Information from the Discovery Endpoint

Now that you confirmed that the project works and that you registered it with Auth0, open its source code in an IDE or text editor (like VS Code or WebStorm) so you can start integrating the application with the OpenID Connect provider. For starters, you will need to make your application issue a request to the Discovery Endpoint to get information from your provider (e.g., where the authorization server resides).

To achieve this, open the `src/server.js` file and search for the line that contains `app.listen` (you can find it at the very bottom of the file). You will need to nest the call to app.listen inside a code that

retrieves information about the OpenID Connect provider. The idea is that, since the app needs this information to enable users to authenticate, you can't let the server listen to users' requests until you get these data:

```
const {OIDC_PROVIDER} = process.env;
const discEnd = 'https://${OIDC_PROVIDER}/.well-known/openid-configuration';
request(discEnd).then((res) => {
 oidcProviderInfo = JSON.parse(res);
 app.listen(3000, () => {
   console.log('Server running on http://localhost:3000');
 });
}).catch((error) => {
 console.error(error);
 console.error('Unable to get OIDC endpoints for ${OIDC_PROVIDER}');
 process.exit(1);
});
```

As you can see, the code that encapsulates app.listen issues an HTTP GET request to a path called `/.well-known/openid-configuration` under the OpenID Connect provider. If the provider fulfills this request correctly, your application will get a string response with the information it will need. Then, to be able to read this information, the application will parse this response into a JavaScript object and save it in a variable called `oidcProviderInfo`. On the other hand, if the application is unable to fetch the provider data, it will log the error and exit with a failure code (`process.exit(1)`).

You might have noticed that the code above uses an environment variable called `OIDC_PROVIDER`. To configure this variable, create a file called .env inside the project root and add the variable to it. Then,

while you are there, add a new environment variable called CLIENT_ID. In the end, the file will look like this:

```
OIDC_PROVIDER=
CLIENT_ID=
```

These are the only two environment variables you will need to integrate your app with the OIDC provider. The first one, OIDC_PROVIDER, will point to the OpenID Connect provider domain. The second one, CLIENT_ID, will contain the Client ID value that Auth0 provides for your regular web application when you register it.

You can find the value for both of them on the page you left open on the Auth0 dashboard. There, you will find a field called Domain, which you can copy the value to add to `OIDC_PROVIDER`, and a field called Client ID, which holds the value you will add to `CLIENT_ID`.

If you are curious about what is the exact data that the application will get back from the provider, you can open `https://${OIDC_PROVIDER}/.well-known/openid-configuration` in a web browser (you will need to replace `${OIDC_PROVIDER}` accordingly). If you do so, you will see information like:

`authorization_endpoint`: The URL where end-users will authenticate

`claims_supported`: An array containing the claims supported (more on that later)

`issuer`: The identifier of the OIDC provider (usually its domain)

`jwks_uri`: Where the provider exposes public keys that can be used to validate tokens

`token_endpoint`: The URL that apps can use to fetch tokens

`userinfo_endpoint`: The URL where apps can learn more about a particular user

Initiating the Authentication Process

Now that your application is acquainted with its OpenID Connect provider, the next thing you will do is to implement the `/login` endpoint. Doing that will allow users to start the authentication process, but the app won't be able to complete it yet. You will cover that in the next section when you implement the `/callback` endpoint.

Back in the `src/server.js` file, search for the `/login` endpoint definition. You will see that, for the moment, this endpoint returns an HTTP status of 501 (Not Implemented). Replace this whole definition with the following code:

```
app.get('/login', (req, res) => {
  // define constants for the authorization request
  const authorizationEndpoint = oidcProviderInfo['authorization_endpoint'];
  const responseType = 'id_token';
  const scope = 'openid';
  const clientID = process.env.CLIENT_ID;
  const redirectUri = 'http://localhost:3000/callback';
  const responseMode = 'form_post';
  const nonce = crypto.randomBytes(16).toString('hex');
  // define a signed cookie containing the nonce value
  const options = {
    maxAge: 1000 * 60 * 15,
    httpOnly: true, // The cookie only accessible by the web server
    signed: true // Indicates if the cookie should be signed
  };
```

```
  // add cookie to the response and issue a 302 redirecting user
  res
    .cookie(nonceCookie, nonce, options)
    .redirect(
      authorizationEndpoint +
      '?response_mode=' + responseMode +
      '&response_type=' + responseType +
      '&scope=' + scope +
      '&client_id=' + clientID +
      '&redirect_uri='+ redirectUri +
      '&nonce='+ nonce
    );
});
```

At the top of the new definition, you are initializing some constants that you will need to create the authorization request:

`authorizationEndpoint`: The authorization URL where you will redirect users

`responseType`: The response type your app expects from the provider

`scope`: The information you want to learn about users authenticating

`clientID`: The identifier that the provider attributes to your app

`redirectUri`: Where the provider will redirect users after the authentication process

`responseMode`: How your application will get the ID Token for the end-user

`nonce`: A random string that helps your app prevent replay attacks

* **Note**: *The text above uses "authorization request" instead of "authentication request" for the same reason OpenID Connect uses Authorization Server to refer to the endpoint where users authenticate (i.e., because this protocol is an extension of OAuth 2.0).*

There is a lot of information packed on the list above. For starters, you are setting the responseType to id_token because the only thing you want to achieve is to have a confirmation that users authenticated successfully and because you want to get this information directly from the authorization process (i.e., you want to implement the Implicit Flow). There a handful of alternatives for this parameter. For example, instead of asking an id_token, you could use code. By doing so, you would be implementing the Authorization Code flow, and your app would only get a code back from the provider.

Now, when it comes to what personal information will learn about the users that are logging in, the constant that matters is the scope one. Since you are just using openid on this parameter, you will only get information that users are logged in and a sub claim with their identifier on the provider. To get more information about users, you would need to add more scopes (for example, profile to get claims like name, family_name, and given_name). You will see this in action soon.

* **Note:** *OpenID Connect uses the term Claims to refer to attributes that carry user information. The authorization server is the entity responsible for issuing these claims. Your app can rely on them because the only two ways to get claims is through a secure channel (HTTPS) or inside ID Tokens. You will learn about token verification and the secure channel in a bit.*

Another important constant you are defining is the responseMode with form_post as its value. This constant will inform the OIDC provider that you want your app to get the response back from it (in this case, the id_token) in the body of an HTTP POST request. The alternatives would be getting the response back as query parameters or in the fragment (which would never reach the server, just the browser). Both alternatives are less secure because they can be either logged by servers or by browsers.

After defining these constants, the endpoint definition is creating another one called options that it will use to configure how the nonce value (the one that prevents replay attacks) will be stored. In this case, the app will append the value to a cookie that can only be used in HTTP requests (that is, invisible for client-side scripts), that is digitally signed, and that will remain valid only for 15 minutes. While validating an ID Token that the provider sends, the /callback endpoint will read the nonce value from cookies so you can compare with what is attached to this token.

Lastly, the new `/login` endpoint is issuing an HTTP 302 (redirect) response to the caller, indicating that they must go to the authorization server with the parameters that the constants above defined. With that in place, if you restart the application, you will be able to click on the login link to check if the redirect is working properly. If everything works as expected, you will end up seeing the login page on your OpenID Connect provider.

Handling the Authentication Callback

After enabling users to initiate the authentication process, the next thing you will have to deal with is the authentication callback. When your users finish dealing with the login page to authenticate, the OpenID Connect provider will redirect them back to the URL you passed on the redirectUri parameter of the authorization request. Since you asked the provider to use form_post as the responseType, the authentication server will generate an ID Token, embed it in an HTML form, and render it on the end-user browser. The page that the provider renders will also include a script that will post the HTML form automatically, as soon as it gets rendered.

The whole process described above is transparent to your users, but will influence how you get ID Tokens.

That is, your server must be able to handle HTTP POST requests and to parse application/x-www-form-urlencoded content type. In the case of the project you are using, you already have the tools needed to parse this type of content. The only thing you will need to do is to search for the /callback endpoint definition and replace it with this:

```
app.post('/callback', async (req, res) => {
  // take nonce from cookie
  const nonce = req.signedCookies[nonceCookie];

  // delete nonce
  delete req.signedCookies[nonceCookie];

  // take ID Token posted by the user
  const {id_token} = req.body;

  // decode token
  const decodedToken = jwt.decode(id_token, {complete: true});

  // get key id
  const kid = decodedToken.header.kid;

  // get public key
  const client = jwksClient({
    jwksUri: oidcProviderInfo['jwks_uri'],
  });
```

```
client.getSigningKey(kid, (err, key) => {
  const signingKey = key.publicKey || key.rsaPublicKey;

  // verify signature & decode token
  const verifiedToken = jwt.verify(id_token, signingKey);

  // check audience, nonce, and expiration time
  const {
    nonce: decodedNonce ,
    aud: audience,
    exp: expirationDate,
    iss: issuer
  } = verifiedToken;
  const currentTime = Math.floor(Date.now() / 1000);
  const expectedAudience = process.env.CLIENT_ID;
  if (audience !== expectedAudience ||
      decodedNonce !== nonce ||
      expirationDate < currentTime ||
      issuer !== oidcProviderInfo['issuer']) {
    // send an unauthorized http status
    return res.status(401).send();
  }

  req.session.decodedIdToken = verifiedToken;
  req.session.idToken = id_token;
```

```
      // send the decoded version of the ID Token
      res.redirect('/profile');
    });
});
```

With these modifications in place, when a user calls this endpoint (as the result of a successful authentication), the new version of it will create a constant that holds the value of the nonce generated for the authorization request. The goal with this constant is to compare it with what is inside the ID Token. Besides that, the endpoint will delete the nonceCookie, so the app doesn't get exposed to replay attacks. Then, the endpoint will read the ID Token sent by the OIDC provider and decode it to be able to see its internal details. With these details in hand, the endpoint starts the process to validate the signature of the ID Token.

Note: Validating the digital signature of ID Tokens is a critical step that your app must do to check if the token was indeed created by the authorization server it trusts. There are some scenarios where this process is not needed, though. For example, when the ID Token is retrieved through a backchannel (i.e., the token doesn't travel through some agent running on users devices), then the application can skip this step and trust the channel (HTTPS) used to get the ID Token.

To be able to verify the signature of ID Tokens, the app will need to get a key from the authorization server. The key that the app will need is the counterpart of the key that the authorization server used to sign the token. These keys are known as the public key (the one your app will use) and the private key (the one that only the authorization server knows). Together, they are used in what is known as an asymmetric algorithm to sign and verify tokens.

To get the public key needed to verify the token signature, the application will need to issue a request to the JWKS URI. This URI will return a set of keys that contains these keys. As you can see on the code, you are using an object called jwksClient to help you with the process of parsing the contents of this URI and to find the right key. All you will need to pass in to create this object is the JWKS URI itself, which you retrieve from the OpenID Connect discovery endpoint (oidcProviderInfo['jwks_uri']) and the kid property that you can find on the header of the ID Token.

With this information, the next thing you do is to call the getSigningKey to retrieve the public key and pass it to the verify method of the jsonwebtoken module. If the ID Token is digitally signed with the counterpart key (i.e., the private one), then the verify method will finish its execution without throwing any error and will send the token back decoded. The contents of the verifiedToken constant will be the same on the decodedToken. The difference is that you can rely on the fact that the former has a valid signature, while on the latter you don't have this guarantee (the only reason you needed it was to get its kid property so you could find the public key).

After this whole dance to verify the signature of ID Tokens, the next thing the application must do is to check four important aspects of these tokens:

`audience:` It is important to confirm that the token was created for this application in particular. So, the expectedAudience (the aud claim) must be the CLIENT_ID defined by the provider when you registered your application there.

`nonce:` As mentioned before, to avoid replay attacks it is also important to confirm that the OpenID Connect provider appended to the ID Token the same nonce value that the application generated when construction the authorization request.

`expiration date:` Another important characteristic of ID Tokens is that the current time

must be before the time represented by the exp claim.

`issuer`: Lastly, but equally important, is that the issuer (the iss claim) of the token must be your OIDC provider.

If any of the checks mentioned above end up being invalid, the /callback endpoint will reject the ID Token and issue a 401 HTTP status back to the end-user. Otherwise, the endpoint will add two information to the end-user session: the decodedIdToken and the original (encoded) id_token.

Adding this information to the session is not necessary and is not something OpenID Connect specifies (or cares about). Actually, after getting the ID Token and verifying it, it is up to the application developers to decide how and when this information will be used. The web application you are dealing with puts this information on the session so it can easily access it later.

After adding this information to the session, the application redirects the end-user to an endpoint called /profile. This endpoint already exists on the initial project and shows some of the information available on the decoded ID Token. Besides that, as you will see, the /profile endpoint also includes a link to https://jwt.io, which is a tool that shows the whole content of JWTs (in this case the id_token).

To see this in action, you can restart the application, open it in your browser, click on the login link, and authenticate with the OpenID Connect provider. When you do so, the provider will send you back to the application with the ID Token. Then, after running the checks mentioned above, the /callback endpoint will redirect you to the /profile page. There, you will be able to see some of the contents of the ID Token and the link to the https://jwt.io tool.

Requesting More Information About Users

After running the application and logging in to it, you probably noticed that the /profile page is missing a lot of data. For example, the profile picture is broken, and there is no email address. The problem here is that your application is not requesting this data when it creates the authorization request. All your application asked for was the openid scope, which makes the provider add only the identifier (the sub claim) of the user to the ID Token and nothing else.

If you want to get more information about who your users are, you have two alternatives. The first one, which you will learn about in this section, is to get a richer ID Token. The second one is by issuing a request to an endpoint called UserInfo, which depends on an access token. You will learn about the second approach later.

To see the first approach in action, open the src/server.js file, and search for the scope constant. For now, this constant contains only the openid value. To get more information about users, you will need to add one of the scopes that the OpenID Connect provider supports. To see what are the scopes your provider supports, you can rely on the Discovery Endpoint once again. If you open https://${OIDC_PROVIDER}/.well-known/openid-configuration in a web browser (don't forget to replace ${OIDC_PROVIDER} accordingly), you will see the scopes_supported property. On this property, you will find an array that contains at least openid (this is mandatory, as described by the OIDC specification) and, probably, other scopes like:

`profile`: A scope that asks access to claims like name, family_name, and birthdate

`email`: A scope that asks access to the email and email_verified claims

`address`: A scope that asks access to the address claim

`phone`: A scope that asks access to the phone and phone_verified claims

So, if you add any of the supported scopes to the scope constant, when your users sign in to your application, you will get more information about them. For example, try adding profile email right next to openid on the scope constant, separating everything with spaces (i.e., in the end, you will have openid profile email as the value of this constant). Then, restart your application and authenticate again. If things work as expected, when you reach the /profile page, you will see the information requested. Besides that, if you click on the https://jwt.io link, you will see a lot more claims in the new ID Token.

Using SDKs to Authenticate Users

How did you find the technical steps needed to handle OIDC authentication in a web application? Although the number of steps is not astronomical, their details are far from simple. Besides that, if you analyze carefully, you will see that the code above takes advantage of some pre-existing packages (like jsonwebtoken) to handle a lot of work related to how OpenID Connect operates. For example, if you would have to learn about token signature verification, you would need an incredible amount of effort only to confirm that a token is correctly signed. That is, you just scratched the surface of the protocol and related technologies on the previous sections.

Having that in mind, in this section, you will learn about an easier way to use OpenID Connect: through SDKs. Here, you will use passport, the most popular authentication middleware for Node.js, and passport-auth0, the official Auth0 library for regular web applications written for this platform. As you will see, things become much simpler when you use an SDK, mainly because you don‘t need to invest time dealing with error-prone technical details that the protocol requires.

To learn how to leverage these SDKs, you will follow steps similar to the ones above. You will fetch a pre-existing project, so you don't have to scaffold one from scratch. Then, you will use some code to make the application integrate with OpenID Connect. As you already registered an app in your Auth0 dashboard, you won't have to do it again.

So, without further ado, open a new terminal, navigate to the directory where you save your projects, and execute the following commands (as an alternative, you can use the green Download button on GitHub and unzip the project locally):

```
git clone https://github.com/auth0-blog/oidc-book-regular-webapp-auth0.git
cd oidc-book-regular-webapp-auth0
```

Then, after loading the code in your preferred IDE, open the src/server.js file, and search for the line that creates the Express server (i.e., search for const app). Right underneath this line, add the following code:

```
// Configure Passport to use Auth0
const auth0Strategy = new Auth0Strategy(
  {
    domain: process.env.OIDC_PROVIDER,
    clientID: process.env.CLIENT_ID,
    clientSecret: process.env.CLIENT_SECRET,
    callbackURL: 'http://localhost:3000/callback'
  },
  (accessToken, refreshToken, extraParams, profile, done) => {
    profile.idToken = extraParams.id_token;
    return done(null, profile);
```

```
  }
);
passport.use(auth0Strategy);
```

The code you are adding creates a new constant called auth0Strategy with the details of your OIDC provider and configures passport to use this constant. There are a few more lines related to configuring passport in your app (like configuring user serialization and initializing it on Express), but they were omitted as they are not that relevant here. What matters is the properties you are using to configure the SDK to integrate the app with your provider and what you do with the information (tokens, mainly) you get from it.

In relation to the properties, there is one major difference between what you have done before and what you are doing now. In this case, it is the introduction of the CLIENT_SECRET property. On the app you built before, you were getting the ID Token directly from the authorization endpoint (i.e., you implemented the implicit flow). As such, you didn't need this property. However, this time, as the passport-auth0 SDK implements the authorization code flow (i.e., your app will get authorization code and exchange them for tokens), you will need to pass in this value as well.

Now, when it comes to the information you get back from the passport-auth0 strategy, you can see that you are only using two things:

`profile` – which is the contents of the payload area of the decoded ID Token

`id_token` – which comes inside the extraParams object

These are the exact same information the other application got at the end of the /callback endpoint. The difference is that, to get this information, you don't need to implement all the details as you have done

before. For example, by leveraging the SDK, you avoided having to deal with:

`Discovery Endpoint integration` – the passport-auth0 SDK will use the domain property passed during Auth0Strategy initialization to fetch this information

`The responseMode value` – the SDK will define that for your application

`The nonce value` – the SDK will also handle that for your app

`The authorization request` – instead of building the request URL by yourself, you can let the SDK handle that for you

To complete this step, create the .env file on the project root. Then, add the following environment variables to it:

```
OIDC_PROVIDER=

CLIENT_ID=

CLIENT_SECRET=
```

Now, open the Settings tab of the application you created earlier in your Auth0 dashboard and use:

`the Domain property` to set the OIDC_PROVIDER variable

`the Client ID property` to set the CLIENT_ID variable

`the Client Secret property` to set CLIENT_SECRET

With that in place, open the src/server.js file one last time, and replace the /login and the /callback endpoint definitions with the following:

```
app.get(
  '/login',
  passport.authenticate('auth0', {
    scope: 'openid email profile'
  }),
);

app.get('/callback', (req, res, next) => {
  passport.authenticate('auth0', (err, user) => {
    if (err) return next(err);
    if (!user) return res.redirect('/login');

    req.logIn(user, function(err) {
      if (err) return next(err);
      res.redirect('/profile');
    });
  })(req, res, next);
});
```

As you can see, for the /login endpoint, all you had to do was to define the information you want to learn about the users (i.e., define the scopes property) and to rely on the SDK (by calling passport. authenticate) to handle the process for you. Now, when a user issues a request to /login, the SDK will create the authorization request and send it to the user for you.

For the /callback endpoint, the code is not that much more complex. In fact, the code for this endpoint calls the same method on the SDK (passport.authenticate) to handle the authentication callback. The difference is that, for this endpoint, you pass a callback function that will define what happens when the OIDC provider sends users back after the authentication process:

- If an error occurs, the code will let the error flow into the next handler. How you will deal with this error is up to you (you can log it, show to users, etc.).
- If there are no errors, the code will call the req.logIn (the logIn method is part of the passport SDK) to set the user on the current session and will redirect users to /profile.

After applying these changes, if you run npm start in your terminal and use the identity provider to log in, you will see that you will get the same page you got at the end of the last section. The difference is that your code now is much leaner. Also, by relying on the official SDK, you can rest assured that you will have an easier life upgrading your code to take advantage of any improvements made by its maintainers.

Recap

This section introduced a good number of concepts that you have to digest. For starters, you learned that the OpenID Connect protocol defines a few different flows that you can leverage to handle end-user authentication. More specifically, you learned about the Implicit flow, which returns token directly from the authorization endpoint; and you learned about the Authorization Code flow, which hands to your app an authorization code instead. You haven‘t had the chance to see the internal details of this last flow yet (since the SDK handled it for you), but you will see that in action soon when you learn how to consume APIs on behalf of your users.

After teaching you about the different flows, this section introduced the Discovery Endpoint. In particular, you learned where to find this endpoint and how it exposes important characteristics of OpenID Connect providers.

Then, you learned how to build authorization requests (which refers to the URL where your end-users authenticate) and how to handle authentication callback. While learning about these topics, you touched important concepts like token verification, redirect URIs, nonce, scope, response type, and more.

Lastly, you had the chance to see how SDKs can facilitate the integration with OpenID Connect providers. With this knowledge, you can consider yourself initiated on the topic, and you are ready to move into more advanced topics and scenarios.

Traditional Web Apps and the Delegated Authorization Flow

So far, you learned how to use OpenID Connect to implement two things in your web applications: an endpoint that creates authorization requests so users can sign in; and another one that handles authorization callbacks to fetch their profile and to complete the authorization requests. With this knowledge, you can consider yourself initiated on the OpenID Connect world and you are ready to start learning about other relevant topics. For instance, one important subject that you still have to delve into is the Delegated Authorization one.

On the OpenID Connect Introduction chapter, you touched this topic superficially. There, you read that OIDC is based on the OAuth 2.0 framework and that this framework was originally created to handle delegated authorization scenarios. Besides that, you read that, to be able to consume information on behalf of its users, applications need access tokens. In this chapter, you will dive a bit deeper into these topics and you will learn what you need to enable your applications to consume information for your users.

The following list illustrates how the rest of the chapter will flow:

- First, you will revisit the definition of the Authorization Code flow.
- Then, you will spin up an API – a Resource Server that contains information that belongs to your users – and you will register it in your Auth0 dashboard so you can protect the resources.
- After that, you will refactor both versions of the application you have been creating so you can learn about each step needed to implement this flow and to turn the app into an OAuth 2.0 Client application.

In the end, you will have an application that enables users to sign in and that can fetch information from the API you spun up on behalf of these users.

The Authorization Code Flow

As the previous chapter introduced, on the Authorization Code Flow, instead of getting tokens directly from the authorization callback request, applications first get authorization codes. Then, they use these codes in a request to another endpoint on the authorization server to exchange them for the tokens they need.

The most significant advantage that this flow has in relation to the implicit flow is its security. More specifically, there are two characteristics of the authorization code flow that help making it a better choice when it comes to security. First, the dance to exchange codes for tokens happen on back channels. That is, instead of having tokens travelling through users devices, the authorization code flow relies on a channel that is open directly between the application and the authorization server. This direct channel drastically decreases the chances of having tokens falling into the wrong hands.

The second characteristic is that, before issuing tokens, authorization servers require applications to authenticate themselves. This authentication process usually happens by applications using credentials that authorization servers assign to them (similar to having a user inputting their usernames and passwords to sign in).

In summary, applications and authorization servers have more confidence in the flow because they use a more reliable channel to exchange tokens and because applications use credentials to authenticate before getting these tokens.

Spinning Up a Resource Server

In a quick recap, Resource Servers refer to the entities that serve resources that belong to users. Just like in the example you read earlier about the Twitter API, a resource server usually refers to an API where user information (resources) are kept safe and accessible only by the users themselves or by third-party applications that were explicitly granted authorization.

In this section, you will spin up an API that will act as a resource server. This API will expose four endpoints that will allow users (or applications on their behalf) to manage to-do items. Each one of these endpoints will have a specific OAuth 2.0 scope associated with it that users will have to grant to applications.

* **Key Term:** *OAuth 2.0 relies on **scopes** to enable Clients (Applications) to inform Resources Owners (Users) what, specifically, they want to do or what they want to access on their behalf. Prior to this chapter, you used scopes on OpenID Connect authorization requests to define what your application wanted to learn about its users. If you think for a bit, the applicability is exactly the same. On OpenID Connect, applications use scopes to define what profile data they want to consume on behalf of its users; on OAuth 2.0, applications use scopes to define what kind of data they want to consume (or action they want to perform) on behalf of these users.*

The scopes that the API will use to restrict access to endpoints will be:

`read:to-dos` - With this scope, applications will be able to read to-do items on behalf of their users.

`create:to-dos` - With this scope, applications will be able to create new to-do items on behalf of their users.

`update:to-dos` - With this scope, applications will be able to update existing to-do items on behalf of their users.

`delete:to-dos` - With this scope, applications will be able to delete existing to-do items on behalf of their users.

Now that you know what the API will be about and that you deepened your knowledge around scopes, you can start configuring the API to run in your local environment. For starters, you will need to head to the APIs sections of your Auth0 dashboard and click on the Create API button. Then, you can fill in the form presented by Auth0 as follows:

API Name: `To-Dos API`

API Identifier: `https://to-dos.somedomain.com`

Signing Algorithm: `RS256`

The name you are adding to your API is not important from the protocol point of view (this is just a field you can use to easily identify what the API is about). However, the other two values are quite important. The first one, the API identifier, is the value that your app will use to tell the authorization server what resource server it needs access to. The second value, RS256, defines the type of algorithm that the authorization server must use to sign access tokens (in this case, you are telling Auth0 to use an asymmetric algorithm).

* **Note:** There is nothing on the OAuth 2.0 framework saying how access tokens should be signed. In fact, access tokens should be considered opaque artifacts that have no structure at all. In other words, applications have to consider them as random characters.

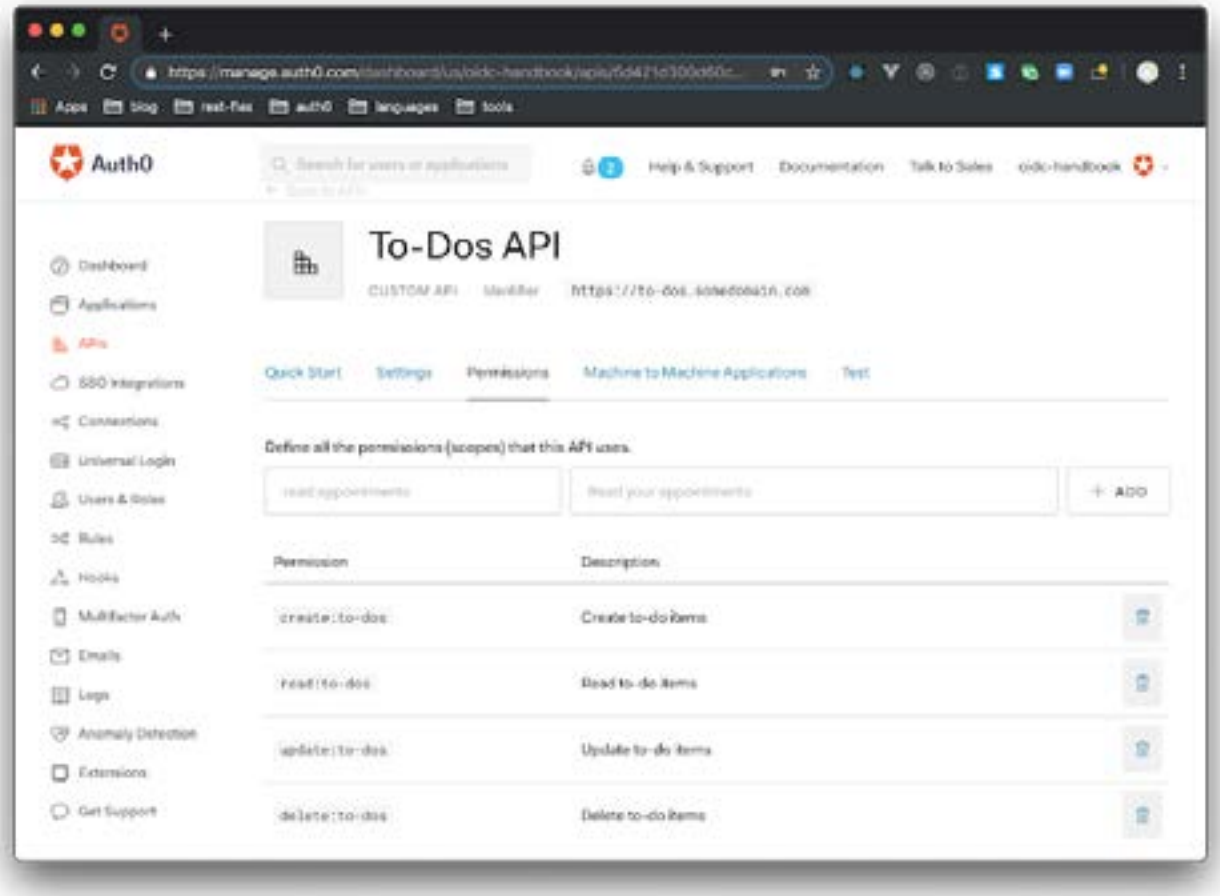
Now, when you click on the Create button, Auth0 will redirect you to the Quick Start section of your new API. From there, head to the Permissions section, and use the form to add the four scopes:

`read:to-dos` - Read to-do items

`create:to-dos` - Create new to-do items

`update:to-dos` - Update existing to-do items

`delete:to-dos` - Remove existing to-do items



There is no save button on this section. So, when you finish inserting these scopes, you can open a terminal to fetch the API project and to run it locally. On this terminal, move to the directory where you usually save your projects and issue the following commands:

```
git clone https://github.com/auth0-blog/oidc-oauth2-api.git

cd oidc-oauth2-api

npm install
```

Instead of executing the first command to clone the API, you can also use GitHub's UI to download the project. Either way, you will have to make sure you move your terminal into the project directory (second command) and that you install all project dependencies (third command). After doing so, you will have to create a new file called .env inside the project root and add the following variables to it:

```
OIDC_PROVIDER=

API_IDENTIFIER=
```

These are the only two environment variables you will need to run this API. For the first variable, you can use the exact same value you used for the `OIDC_PROVIDER` variable on the traditional web application (i.e., something similar to `oidc-handbook.auth0.com`). For the second variable, you will have to use the identifier of the API you just registered in your Auth0 dashboard (i.e., something like `https://to-dos.somedomain.com`). With that in place, you can run `npm start` to trigger the API. After running it, you can issue some HTTP requests to it to confirm that the access to the API is indeed restricted:

```
curl -I http://localhost:3001/
```

If you issue this command, you will see that the API responds with HTTP 401 Unauthorized. At the end of this chapter, you will see the API responding with to-do items back to requests issued by your application.

Requesting Delegated Authorization

After spinning up the resource server, it is time to refactor the application you built on the last chapter (the one that does not use Auth0's SDK) to make it request delegated authorization to the To-Do API when users sign in. That is, in this section, you will refactor the `/login` endpoint to make it trigger an authorization code flow that informs the authorization server that the application wants to perform some actions on behalf of its users; and, you will refactor the `/callback` endpoint to exchange codes for access tokens (and ID tokens as you still want to sign users in to your app).

To start this refactoring, open the `src/server.js` file, go to the `/login` endpoint definition and change three of the constants defined on the top of this endpoint:

`responseType`: Change this constant to `code` to make the authorization request inform the authorization server that the application wants to use the authorization code flow.

`scope`: Add `read:to-dos` to the list of scopes the application is requesting. This scope will let the authorization server know what kind of action the application wants to execute on behalf of its users.

`responseMode`: Change this constant to `query` to inform the authorization server that the application wants to get the `code` as a query parameter.

* **Note:** *Getting codes as query parameters is not a problem because they contain no personal information about users or any other sensitive data (whereas ID tokens and access tokens do). Even if someone intercepts this code, they would also need to know the application credentials to be able to exchange it for tokens.*

After changing these constants, add another one right after the nonce constant:

```
const audience = process.env.API_IDENTIFIER;
```

Then, use the new constant on the call to the `redirect` function to let the authorization server know that your application wants to get access to the `API_IDENTIFIER` audience. After these changes, the `/login` endpoint definition will look like this:

```
app.get('/login', (req, res) => {
 // define constants for the authorization request
 const authorizationEndpoint = oidcProviderInfo['authorization_endpoint'];
 const responseType = 'code';
 const scope = 'openid profile email read:to-dos';
 const clientID = process.env.CLIENT_ID;
 const redirectUri = 'http://localhost:3000/callback';
 const responseMode = 'query';
 const nonce = crypto.randomBytes(16).toString('hex');
```

```
const audience = process.env.API_IDENTIFIER;

// define a signed cookie containing the nonce value
const options = {
   maxAge: 1000 * 60 * 15,
   httpOnly: true, // The cookie only accessible by the web server
   signed: true // Indicates if the cookie should be signed
};

// add cookie to the response and issue a 302 redirecting user
res
.cookie(nonceCookie, nonce, options)
.redirect(
authorizationEndpoint +
    '?response_mode='+ responseMode +
    '&response_type='+ responseType +
    '&scope='+ scope +
    '&client_id='+ clientID +
    '&redirect_uri='+ redirectUri +
    '&nonce='+ nonce +
    '&audience=' + audience
  );
});
```

As you can imagine, you still need to define the **API_IDENTIFIER** variable in your application. To do so, open the **.env** file and add the variable there:

```
# … other variables …

API_IDENTIFIER=
```

To configure this variable, you will have to use the identifier you gave for the API you registered in your Auth0 dashboard (i.e., the same value you set on the **API_IDENTIFIER** of the API project).

Then, the next thing you will do is to create a new function called v**alidateIDToken** right after the /login endpoint definition. This function will do a job similar to what the current version of the **/callback** endpoint is doing. That is, it will validate ID tokens, but in a different way:

```
function validateIDToken(idToken, nonce) {
 const decodedToken = jwt.decode(idToken);

// fetch ID token details
 const {
   nonce: decodedNonce,
   aud: audience,
   exp: expirationDate,
   iss: issuer
 } = decodedToken;
 const currentTime = Math.floor(Date.now() / 1000);
 const expectedAudience = process.env.CLIENT_ID;
```

```
// validate ID tokens
 if (
   audience !== expectedAudience ||
   decodedNonce !== nonce ||
   expirationDate < currentTime ||
   issuer !== oidcProviderInfo['issuer']
 )
   throw Error();

 // return the decoded token
 return decodedToken;
}
```

If you compare the code of this function with the one inside **`/callback`**, you will see that the main difference is that the new function does not validate the signature of ID tokens. It wouldn't be a problem to check ID tokens signatures. However, it wouldn't be useful too. The thing is, since the new version of the application will get these tokens through a secure channel (which uses TLS certificates), the app can rely on the integrity of the information. In the end, after validating ID tokens, the new function will return the decoded version of them so your application can keep showing profile information about users.

> ✱ **Note:** *The application still needs to check the other characteristics of the ID tokens before using them (i.e., the audience must be the application itself, tokens can't be expired, and so on).*

Now that you have a function to validate ID tokens, you can refactor the /callback endpoint. So, find this endpoint definition and replace it with the following:

```
app.get('/callback', async (req, res) => {
 const { code } = req.query;

const codeExchangeOptions = {
   grant_type: 'authorization_code',
   client_id: process.env.CLIENT_ID,
   client_secret: process.env.CLIENT_SECRET,
   code: code,
   redirect_uri: 'http://localhost:3000/callback'
 };

 const codeExchangeResponse = await request.post(
   'https://${process.env.OIDC_PROVIDER}/oauth/token',
   { form: codeExchangeOptions }
 );

 // parse response to get tokens
 const tokens = JSON.parse(codeExchangeResponse);
 req.session.accessToken = tokens.access_token;

 // extract nonce from cookie
 const nonce = req.signedCookies[nonceCookie];
 delete req.signedCookies[nonceCookie];
```

```
try {
   req.session.decodedIdToken = validateIDToken(tokens.id_token, nonce);
   req.session.idToken = tokens.id_token;
   res.redirect('/profile');
 } catch (error) {
   res.status(401).send();
 }
});
```

There are a good number of differences between this version of the `/callback` endpoint and the old one. For starters, the endpoint now accepts HTTP GET requests (instead of POST). This change is needed because the `query` response mode makes the authorization server redirect users back to your application through a GET request instead of an automated form submission.

Another difference is that, instead of focusing on ID token validation (which is now handled by the `validateIDToken` function), this endpoint is more focused on exchanging the code retrieved from the authorization server with the tokens the application needs (in this case, ID tokens and access tokens). As you can see on the `codeExchangeOptions` constant and on the `request.post` method call, the `/callback` endpoint performs these exchanges by issuing POST requests to the `/oauth/token` endpoint on the authorization server with five parameters:

`grant_type`: This parameter uses the `authorization_code` value to make it explicit to the authorization server what flow the application is using.

`client_id and client_secret`: These parameters are the credentials of the application, so the authorization server knows who it is talking to.

`code`: This is the **`code`** the application gets back from a successful authorization request.

`redirect_uri`: Since this parameter was included in the authorization request, the OAuth 2.0 framework requires applications to include on requests for code exchange.

After issuing the code exchange request and getting a response from the authorization server, the **`/callback`** endpoint adds the access token to the user session and calls the **`validateIDToken`** to complete the process. In the end, the endpoint redirects users to the **`/profile`** endpoint, just like before.

After completing the authorization code flow to fetch tokens from the authorization server, the last thing you will do is to see how to use access tokens to consume resources (or execute actions) on behalf of your users. To see this, search for the /to-dos endpoint definition and replace it with this:

```
app.get('/to-dos', async (req, res) => {
 const delegatedRequestOptions = {
   url: 'http://localhost:3001',
   headers: {
     Authorization: 'Bearer ${req.session.accessToken}'
   }
 };
 try {
   const delegatedResponse = await request(delegatedRequestOptions);
   const toDos = JSON.parse(delegatedResponse);

res.render('to-dos', {
     toDos,
   });
```

```
  } catch (error) {
    res.status(error.statusCode).send(error);
  }
```

Note how the new version of this endpoint is configured to issue HTTP requests to the API you spun up before (`url:'http://localhost:3001'`) with the access token of the user (`req.session.accessToken`). More specifically, these requests append access tokens to the Authorization header with the `Bearer` prefix. This approach turns tokens into bearer tokens, which is defined on the Bearer Token Usage specification.

After issuing the request, if the API accepts the token (which will only happen if they get valid access tokens that include the `scope` they are expecting), your application will get whatever it requested for and will be able to use the information as needed. In this case, the app will get a list of to-do items (`const toDos`); then, it will use this information to render a page with all the items saved on the API.

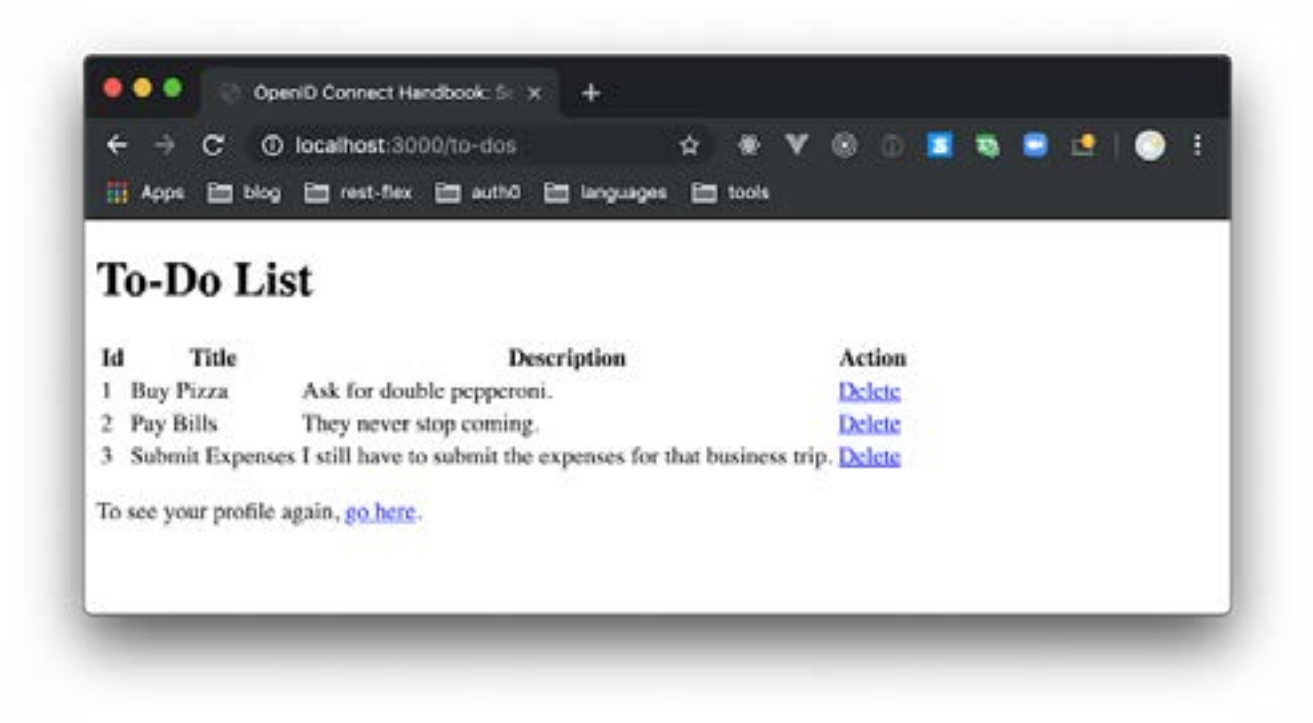
Now, before being able to restart your web application, the last thing you will have to do is to define the `CLIENT_SECRET` environment variable. To do so, open the `.env` file and insert this variable:

```
# … leave the other environment variables untouched ...

CLIENT_SECRET=
```

Then, open the application on your Auth0 dashboard, copy its Client Secret property, and use it to configure the new variable. You can also copy this value from the `.env` file of the other version of the web application (the one that uses the SDK).

At this moment, if you restart the web application (you can hit `Ctrl + C` and then `npm start` on its terminal), after signing in, you will get access to the `http://localhost:3000/to-dos` page.



The information you see on this page is the information that the To-Do API is providing to the application. That is, the three items on the to-do list you see in your browser are the information the application consumed from the API on your behalf.

* **Note:** *If you use different users to sign in to the application, you will notice that the To-Do API always returns the same data. The API behaves like that to keep the instructions in this book simpler. In a real resource server API, normally, different users would have different data.*

After implementing the `/to-dos` endpoint and testing the application, there is one more interesting thing you can do. You can try implementing the `/remove-to-do/:id` endpoint as an exercise. The code to implement this endpoint will be quite similar to the /to-dos endpoint, but (to give you a hint) you will

need an access token that is a bit different and you will need to fetch the `:id` value from requests.

Using SDKs to Request Delegated Authorization

Just like in the Using SDKs to Authenticate Users section of the last chapter, this section will show you how much simpler things can be when leveraging SDKs. In fact, Auth0's SDKs make things so easy that, to make your application request delegated authorization, all you have to do is to add two lines of code to the project that uses the SDK and change another one.

To see this in action, load the project that uses Auth0's SDK in your IDE, then open the **`src/server.js`** file. Inside this file, find the **`auth0Strategy`** constant definition and add the access token to the profile of the user. After doing so, the **`auth0Strategy`** constant will look like this:

```
// Configure Passport to use Auth0
const auth0Strategy = new Auth0Strategy(
 {
   domain: process.env.OIDC_PROVIDER,
   clientID: process.env.CLIENT_ID,
   clientSecret: process.env.CLIENT_SECRET,
   callbackURL: 'http://localhost:3000/callback'
 },
 (accessToken, refreshToken, extraParams, profile, done) => {
   profile.idToken = extraParams.id_token;
   profile.accessToken = extraParams.access_token;
```

```
    return done(null, profile);
  }
);
```

This is the first line you had to add to this project. Now, to make the other two changes, find the **`/login`** endpoint definition and replace it with this:

```
app.get(
  '/login',
  passport.authenticate('auth0', {
    audience: process.env.API_IDENTIFIER,
    scope: 'openid email profile read:to-dos'
  })
);
```

As you can see, the new definition is adding the **`audience`** parameter with the **`API_IDENTIFIER`** to the object passed to the **`passport.authenticate`** call and adding **`read:to-dos`** to the list of scopes the application will request. With these changes, Auth0's SDK will be able to inform the authorization server what type of access token the application needs (i.e., to what audience and with what scopes) and your app will be able to consume the To-Do API on behalf of its users.

Then, to complete this feature is to replace the definition of the **`/to-dos`** endpoint:

```
app.get('/to-dos', async (req, res) => {
  const delegatedRequestOptions = {
    url: 'http://localhost:3001',
```

```
    headers: {
      Authorization: 'Bearer ${req.session.passport.user.accessToken}'
    }
  };
  try {
    const delegatedResponse = await request(delegatedRequestOptions);
    const toDos = JSON.parse(delegatedResponse);
    res.render('to-dos', {
      toDos
    });
  } catch (error) {
    res.status(error.statusCode).send(error);
  }
});
```

The only difference between the definition of this endpoint and the one on the project that does not use Auth0's SDK is from where the application gets users' access tokens. The rest remains the same.

To run the new version of this application, open the `.env` file, add the `API_IDENTIFIER` variable passing the ID of your Auth0 API to it (i.e., the same value you used on the last section), and run `npm start`. You will see that, with just these small changes, you will achieve the same result as before.

Recap

In this chapter, you took a deeper dive into the delegated authorization concept and you read about how the Authorization Code flow helps applications to get access tokens in a secure way. That is, you learned that, by using back channels and application credentials, this flow avoids sending tokens through users devices, which ends up enhancing the security of the whole process.

After the abstract introduction to these topics, you spun up an API (a resource server) while digging deeper into the concept of OAuth 2.0 scopes. Then, you refactored both versions of your web application (the one without Auth0's SDK and the other one that leverages this SDK) and you saw, in action, how to implement the Authorization Code flow and how applications can consume resource servers on behalf of their users.

Step by step, you are building a deeper understanding of OpenID Connect and the OAuth 2.0 framework. In no time, you will see that you will start feeling more comfortable around the concepts of these specifications and you will see that, although complex, using OpenID Connect and OAuth 2.0 is not that hard, mainly if you use reliable SDKs.