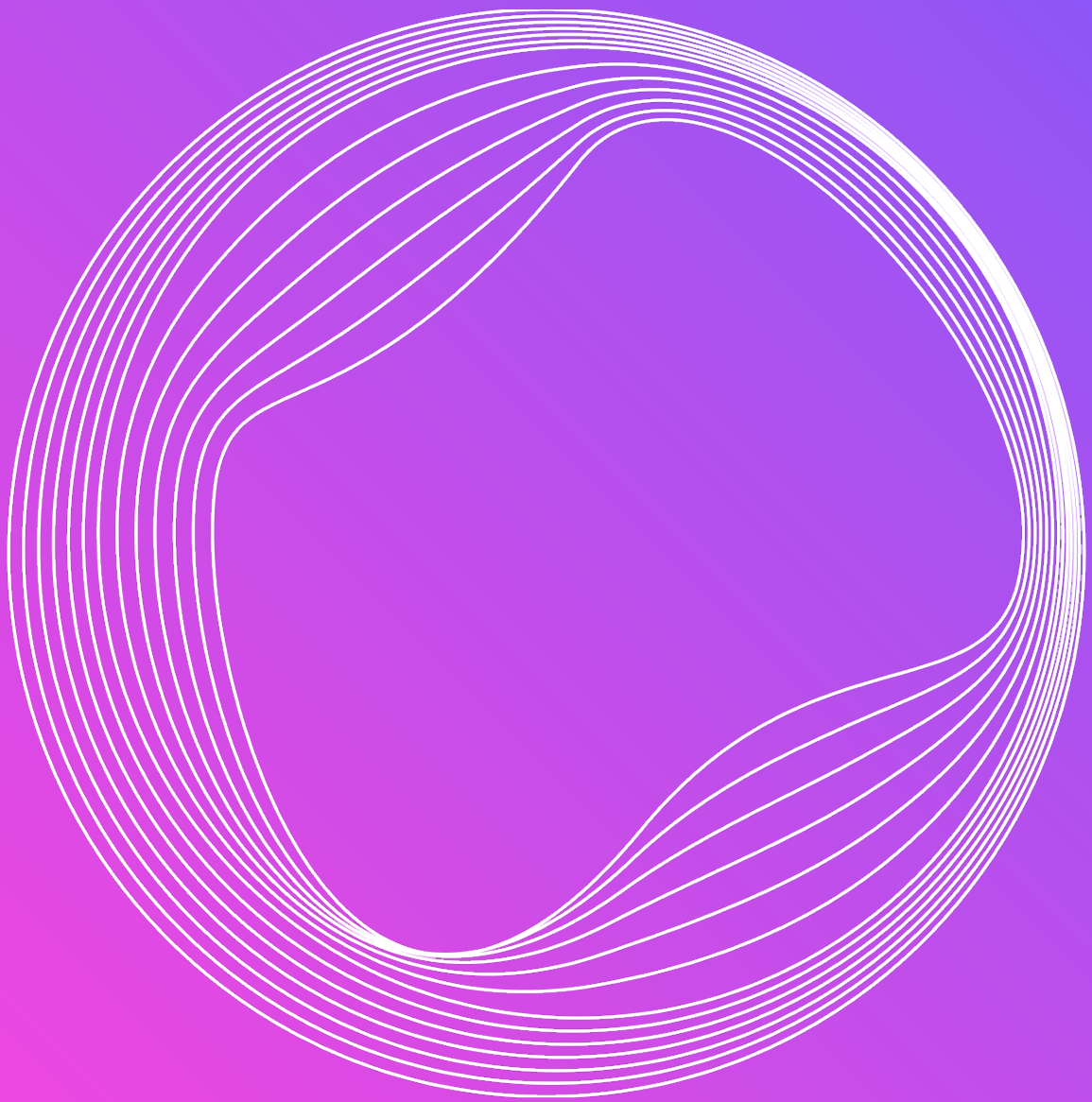
auth0

.NET Identity with Auth0

by Andrea Chiarelli



EBOOK

auth0.com

Contents

auth0.com | © Auth0 2022

Introduction

The recent evolution of .NET aims to provide one development platform for any kind of application and operating system. Starting with .NET 5.0, Microsoft's ambitious project has become a reality: now developers can build web, desktop, and mobile applications for Windows, Mac, Linux, Android, and iOS operating systems leveraging the .NET unified platform.

However, the .NET platform offers more than the ability to create different types of applications for different operating systems. It offers developers a unified programming model and APIs to address common problems. In other words, it improves the developer experience by sharing concepts and preserving knowledge when moving from one project type to another, regardless of the target operating system. The power of this unified development platform can be further boosted with a neutral, standards-based, and scalable platform to manage identity: Auth0.

While the .NET platform aims to relieve the developer's burden in building applications, Auth0 helps make them more secure by simplifying Identity and Access Management (IAM). Auth0 offers authentication and authorization services out of the box, allowing developers to focus on the business logic of their applications. It provides standards-based services that integrate with any programming language and development framework.

How can we combine the power of the .NET platform with the flexibility of Auth0?

This is the goal of this book: to show how you can leverage Auth0's authentication and authorization services in the various application types you can create with .NET. Throughout the book, you will be guided in the steps for adding Auth0 to your applications with a practical approach. You will build and integrate simple applications to learn by doing and will have the opportunity to download their code to play with.

Have fun!

auth0.com | © Auth0 2022

Prerequisites

.NET is a large ecosystem with a 20-year history. Many things have happened during this time — among the others, a few renamings. So, you may have heard about .NET Framework, .NET Core, and .NET (not mentioning .NET Standard). If you want to learn more about the differences between them, check out the article **.NET Core vs .NET Framework vs .NET Standard: A Guided Tour**.

In this book, we will take as our reference .NET 6.0. This means that, while Auth0 supports previous versions of the .NET ecosystem, the code samples provided in this book work on .NET 6.0 and subsequent versions. So, to build the sample applications described in this book, you need .NET 6.0 SDK or above. To check if you have the correct version installed on your machine, type the following command in a terminal window:

```
dotnet --version
```

If you don't have the correct version, download the **.NET SDK** and install it on your machine.

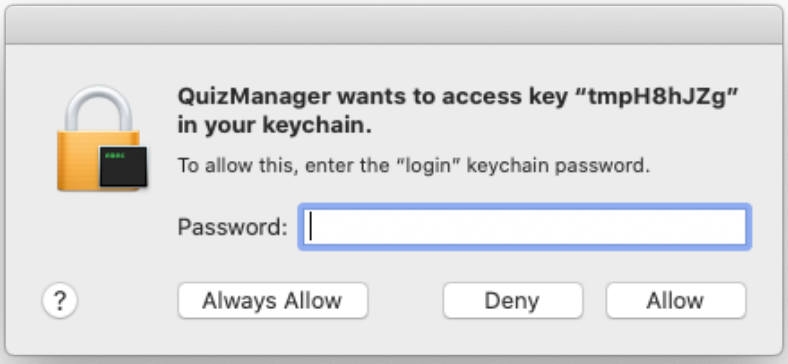
The book will use the **.NET CLI** to manage, build, and run the .NET projects from the command line. The .NET CLI is included with the .NET SDK, so you don't need to install anything else. This choice makes the book independent of your machine's operating system. In addition, by using the .NET CLI, you are not tied to a particular editor or IDE.

Alternatively, you can use Visual Studio, Visual Studio Code, or JetBrains Rider. If you want to use Visual Studio, you should be aware that .NET 6.0 is supported by Visual Studio 2022 and Visual Studio 2022 for Mac. Also, they include the .NET SDK, so you don't need to download it separately.

auth0.com | © Auth0 2022

Heads up for Mac users!

If you are using a Mac, you may be affected by an issue when running an ASP.NET Core application through the .NET CLI. The following window may appear repeatedly:



To work around the issue, open your ASP.NET Core project file (e.g., `MyProject.csproj`) and add the `<UseAppHost>false</UseAppHost>` element to the `<PropertyGroup>` element.

The following example shows what the content of the project file should look like:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>
  <TargetFramework>net6.0</TargetFramework>
  <Nullable>enable</Nullable>
  <ImplicitUsings>enable</ImplicitUsings>
  <UseAppHost>false</UseAppHost>
</PropertyGroup>

</Project>
```

auth0.com | © Auth0 2022

Visual Studio users are not affected by this issue.

Development certificate

The very first time you run an ASP.NET Core application, you should trust the HTTPS development certificate included in the .NET Core SDK. This task depends on your operating system. Please take a look at the **official documentation** to follow the proper procedure.

The Book's Organization

A general introduction to identity concepts and the Auth0 features that you will use precede the hands-on chapters that show how to integrate Auth0 with the various .NET application types.

In **Chapter 1 - Authentication and Authorization**, you will learn the basic concepts of identity and access management and the standards you will use under the hood.

In **Chapter 2 - Introduction to Auth0**, you will be introduced to the main Auth0 features and terminology you will use while integrating your application.

In **Chapter 3 - Regular Web Applications**, you will explore how to integrate Auth0 in ASP.NET Core MVC, Razor Pages, and Blazor Server frameworks.

In **Chapter 4 - APIs**, you will familiarize yourself with protecting ASP.NET Core Web API and Minimal Web API with Auth0.

In **Chapter 5 - Single-Page Applications**, you will discover how to integrate Auth0 with Blazor WebAssembly applications.

Finally, in **Conclusion**, you will recap what you learned and get directions for making your Auth0 integration more effective.

auth0.com | © Auth0 2022

Chapter 1 - Authentication and Authorization

auth0.com | © Auth0 2022

Before we get to work on the code, let's introduce a few concepts that will come in handy when dealing with the integration between an application and Auth0. You may already know some of them, but it's worth spending a few minutes to be sure that we are on the same page.

Specifically, we will discuss a few fundamental concepts about identity and access management, such as authentication and authorization, and the two main standards for modern identity: OpenID Connect and OAuth 2.0. Knowing these concepts will help you better understand how the integration between your app and Auth0 works.

Authentication

Authentication is probably the identity concept best known to developers since it's the feature that opens the door of their application. We can define it as follows:

> **Authentication** is the process of verifying the identity of a user or application (entity). It allows you to validate that someone is who they claim to be.

The way the identity of an entity is verified is what distinguishes the different authentication methods. Think of the username and password you provide to access an online service. These **credentials**, i.e., your username and password, allow the authentication service to verify with a certain level of assurance that the user attempting access is actually you. The credentials you provide to the authentication service are the authentication factor it relies on to verify your identity — that is, to authenticate you.

> An **authentication factor** is a category of credentials that is intended to authenticate an entity.

Although username and password are the most well-known authentication factor, it is not the only one. Typically, authentication factors are grouped into at least the following three categories:

- **Something you know.** This category of authentication factors relies on information that only the user knows. Passwords fall into this category.
- **Something you have.** Authentication factors in this category are objects, services, or similar items that the user can demonstrate they possess. A typical example is the user's smartphone. Another example is access to a mailbox.
- **Something you are.** These factors are strictly tied to a physical characteristic of the user (*biometric factors*), such as their fingerprint, voice, or face.

As you may have guessed, these authentication factor categories have different levels of security. For example, a password is not difficult to steal and can be done without the user's awareness. Stealing a user's smartphone is a bit harder, and the user will notices it in a short time. Even harder is to take away or otherwise imitate a user's physical characteristics.

The user authentication process must balance usability with security. While setting up a password is easy for the user, it's not so secure. On the other hand, relying only on the user's smartphone for authentication may be a risk in case they lose it. Using multiple authentication factors is the best practice to protect users' identities.

> **Multi-factor authentication (MFA)** is an authentication method that relies on multiple authentication factors.

The combination of username and password and a notification on your smartphone is an example of *two-factor authentication*, i.e., authentication based on two factors: your username and password on one side and your smartphone on the other.

Auth0 offers traditional username and password authentication as well as MFA out of the box, in addition to other security services, to make your users' experience as smooth as possible while protecting their identities.

auth0.com | © Auth0 2022

Authorization

We can define authorization as follows:

> **Authorization** is the process of giving a user or an application (entity) permission to access a resource.

The concepts of authentication and authorization are confused by developers more often than you think. While they are usually related, they are independent of each other conceptually. Perhaps the confusion originates from the fact that we very often give a user permission to access a resource based on their identity. In other words, we base our authorization process on the result of authentication. While this is usually the case, it is not always true.

Think of when you get on the bus. No one asks to know your identity. All you need to be authorized to board is a ticket. In this case, the authorization process is not tied to the authentication process.

In general, we say that the authorization process checks some rules to determine whether an entity is allowed to access a resource. We call those rules authorization policies.

> An **authorization** policy is a set of requirements that users or applications must meet to be allowed to access a resource.

Permissions and Privileges

A typical approach to defining authorization policies is based on permissions and privileges.

> A **permission** is a declaration of an action that can be performed on a resource.

Let's consider a resource such as a file. Reading, writing, and deleting that file are permissions — that is, they are operations that can be executed on the file.

auth0.com | © Auth0 2022

> A **privilege** is a permission assigned to a user or application.

When a user is assigned permission to read that file, we say that they have reading privileges for that file.

An authorization policy based on permissions and privileges checks whether a user or application trying to access a resource has the required privileges.

> Although there is technically a difference between permissions and privileges, commonly the two terms are used interchangeably.

Role-Based Access Control

Permissions can be assigned to users to allow them to perform specific operations on a resource. While this seems an easy task, it may soon become challenging when the number of permissions, users, and resources grow; more importantly, users' privileges can change over time.

Consider a company payroll system. Different people accessing this system have different privileges to view and change data. Each employee can view their own data, while HR people can view all employees' data. Also, while an employee can only view their own data, HR people can change them. Also, the HR department may have just a few people who should have the ability to add new employees to the payroll system, with the remainder only having the privilege to update existing employees.

Assigning specific permissions individually to each employee can be challenging and lead to mistakes. To expand on the payroll system example, consider that we have at least four types of permissions. Each of these must be assigned with perfect accuracy to each employee in your company. Even with just a handful of employees and a few HR events a month, this can quickly spiral into a time-consuming and error-prone activity. This doesn't even consider the need to reassign permissions for a large group of people in case of a security policy change.

auth0.com | © Auth0 2022

Role-Based Access Control (RBAC) helps with permission assignment by introducing the concept of roles.

> A **role** is a collection of permissions that can be assigned to users.

Basically, you build a predefined set of permissions, give it a name, such as *Employee*, *HR Assistant*, or *HR Manager*, and assign that role to a user. If you need to add or remove a permission for all the users who have a specific role, you just need to add or remove that permission for the role they are assigned. That's a great improvement!

Other authorization models

In addition to authorization based on permission and role checks, other authorization policies exist. We will not go through all of them, but will mention a few just to make you aware of the complexity behind the authorization process.

Attribute-Based Access Control (ABAC) is an authorization model that evaluates multiple attributes rather than just roles and permissions to grant access to a resource. Those attributes are related to several aspects. For example, they can be related to the user (their role, the organization and/or the team they belong to, etc.), the environment (time of day, current location, etc.), the resource (creation date, data sensitivity, etc.), the action to perform (read, write, etc.), and so on.

In this context, an authorization policy can be something like the following:

An accountant can create an invoice only during working hours.

As you can see, the ABAC model allows you to define very specific authorization policies.

The **Relationship-Based Access Control** (ReBAC) model allows the definition of authorization policies based on the relationships between entities, such as users, resources, applications, etc. We can build

auth0.com | © Auth0 2022

relationships between users and resources but also between users or between resources. For example, in this model you can define relationships as the following:

- John is Sam's manager
- A manager is the approver of an expense report
- An employee is the submitter of an expense report
- John is the submitter of expense report no. 541

Authorization decisions are made based on the context defined by these relationships, which can be as complex as desired and can dynamically change.

As you can see, permissions and roles solve a limited set of authorization scenarios compared to ABAC and ReBAC models.

OpenID Connect and OAuth2

Two open standards represent what is commonly called "modern identity": **OpenID Connect** and **OAuth**. Both standards are closely related but have different goals in their original intent. OpenID Connect is focused on authentication; OAuth is focused on authorization. They differ from previous technologies such as **Security Assertion Markup Language** (SAML) in their flexibility and lower complexity. Both have been largely adopted in the software industry, and many identity providers, such as Auth0, support them.

Let's go into some detail to understand what it is all about.

OAuth

OAuth is an open standard for delegated authorization.

auth0.com | © Auth0 2022

> **Delegated authorization** is the process of giving a service or application permission to access a user's resource on their behalf.

As an example, consider when you give LinkedIn the ability to publish your post on Twitter as well. In this case, you authorize LinkedIn to access your Twitter profile and post a tweet on your behalf. In other words, you delegate LinkedIn to tweet for you.

Before OAuth, the typical procedure for this kind of operation was simply sharing your password with the application that would access your resource, i.e., sharing your Twitter password with LinkedIn, to stick with the previous example. This approach raises many security concerns. First, sharing a password is a well-known bad practice, because you can easily lose control of it. Second, sharing your password allows the application to *impersonate* you — that is, the application can do everything you can on that resource. In the LinkedIn and Twitter example, LinkedIn would be able not only to post tweets but also delete them, change your profile, and even change your password!

OAuth solves this problem by proposing a clear architecture and some authorization flows for specific scenarios. We will not go through the details of the OAuth framework, but learning its terminology will be useful in understanding how to integrate Auth0 into your .NET applications.

Be aware that the current version of OAuth specifications is 2.0. You will usually see it referred to as OAuth 2.0 or OAuth2. A new OAuth 2.1 version is in progress and will consolidate some security practices.

OAuth actors and artifacts

OAuth 2.0 defines four actors involved in an authorization transaction:

- The **resource owner**. This is typically the user, the person who has full permission for a resource.

- The **client**. The client is the application that wants to access the user's resource. In the LinkedIn and Twitter example above, LinkedIn is the client.

- The **resource server**. This is the application that controls access to the resource. Usually, this application is accessible by the client through an API. In our example, Twitter is the resource server. It controls access to your profile and can be accessed by LinkedIn through its API.

- The **authorization server**. The authorization server is the intermediary between the above actors. It receives authorization requests from the client, asks for approval from the user, shares with the resource server details about the authorization process, and so on.

In the interaction between the OAuth actors, three elements are very important:

- The **consent screen**. When the authorization server receives a request to access a resource from the client, it asks the user for confirmation. It shows a screen with the details of the request: the requesting application, the requested resource, and the requested operations on that resource. The user has the ability to grant or deny that access.

- The **access token**. Once the user grants access to the client application, the authorization server provides it with an access token. This is a string that authorizes the client to access the resource on behalf of the user. While the format of the access token is not defined by OAuth specifications, a widely adopted format is **JSON Web Token** (JWT). Access tokens issued by Auth0 are in JWT format.

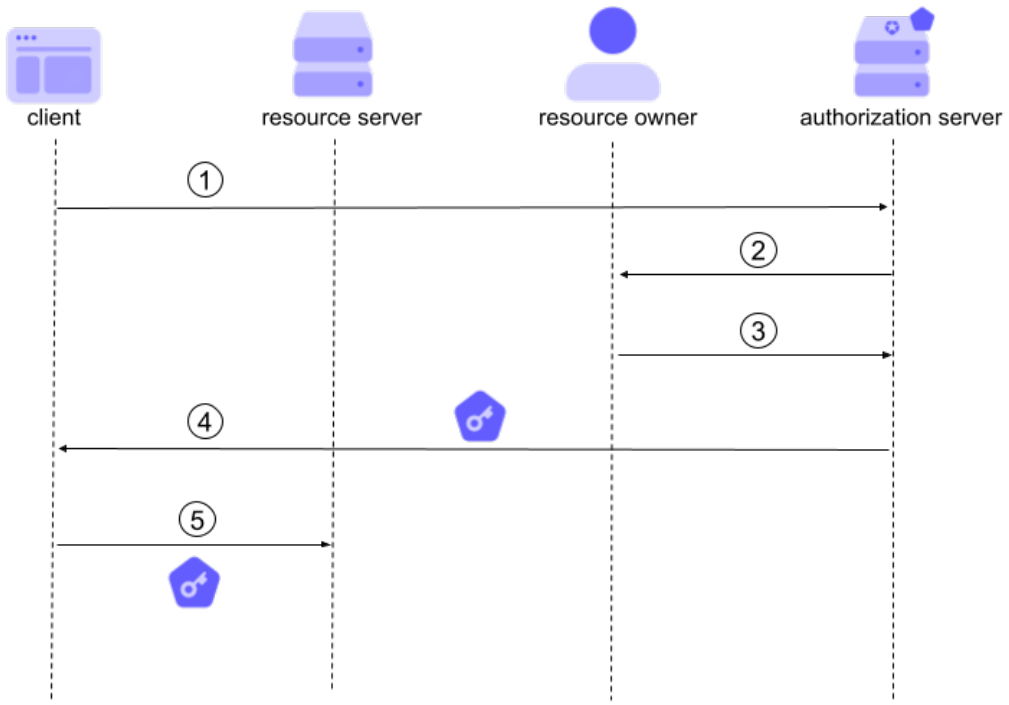
- The **scopes**. Unlike impersonation, delegated authorization allows the user to restrict the set of operations that a client can perform on a resource. In fact, a client can express the operations to perform on the resource through scopes. Scopes are strings that represent operations that can be performed on a resource. They are similar to permissions but have a different semantic in the delegated authorization context.

auth0.com | © Auth0 2022

OAuth flows

Now that you've met the actors and some artifacts provided by OAuth, let's learn how all of them interact.

Consider the following diagram:



It represents a generic flow to authorize the client to access a resource on behalf of the user. Let's follow each step of this flow through the numbered arrows:

1. The client contacts the authorization server and requests access to the resource controlled by the resource server.
2. The authorization server asks for the user's consent to grant access to the resource. Here is where the consent screen goes into action.
3. The user gives their consent to the authorization server.
4. The authorization server issues an access token and provides it to the client.

auth0.com | © Auth0 2022

5. The client uses the access token to access the resource through the resource server.

This is a simplified version of a basic OAuth flow. The real flow has a few more details that we omitted for simplicity. Actually, the flow shown in the diagram is an ideal flow, where the most critical step is step 4: the access token delivery. Most of the additional details in a real flow have to do with security concerns, and in fact, security is the main reason OAuth has different flows. Each OAuth flow is based on a specific scenario and application type. Let's describe a few of them:

- **Authorization Code Flow**. This flow is usually used with regular web applications. It relies on the exchange of an *authorization code* for the access token between the client and the authorization server.
- **Authorization Code Flow with PKCE**. This flow is similar to the Authorization Code Flow but has one more security measure: the use of a *Proof Key for Code Exchange* (PKCE) — that is, a piece of information that makes sure the receiver of the token is the original requester. This flow is used when the client is a Single-Page Application (SPA) or a native application.
- **Client Credentials**. In this flow, the client requests an access token using its own credentials. Actually, this is not a delegated authorization flow, i.e., the client requests to access a resource on its behalf, not on the user's behalf. This flow is used in scenarios with no user interactions, such as in a machine-to-machine scenario or a microservice environment.

OpenID Connect

OpenID Connect (OIDC) is an authentication protocol built on top of OAuth 2.0. It allows applications to verify the identity of a user and optionally receive basic profile information. When the user authenticates, the

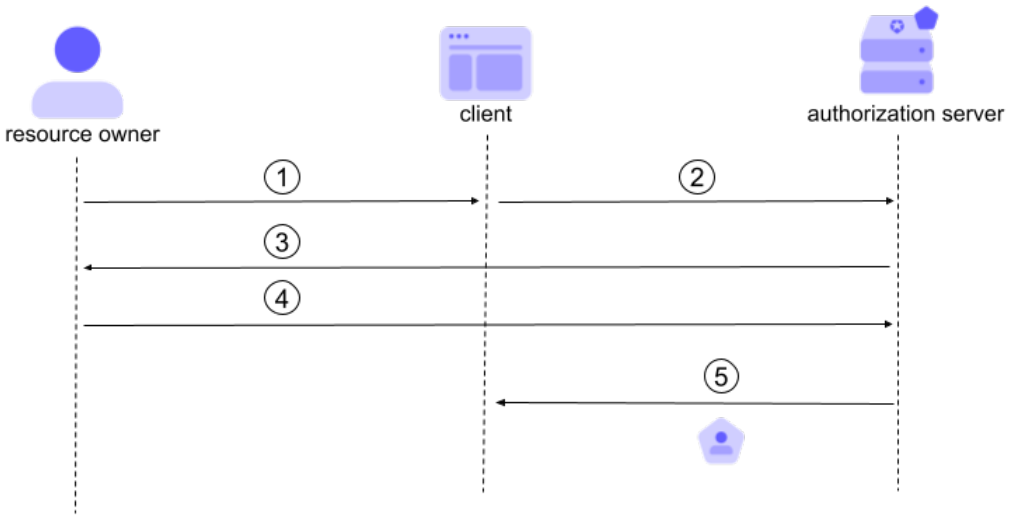
auth0.com | © Auth0 2022

authorization server sends the client an **ID token**, an artifact in JWT format that proves that the user has been authenticated. The user's profile data can be stored in the same token in the form of **claims**, as shown in the following example of a decoded ID token:

```
{
  "iss": "http://my-domain.auth0.com",
  "sub": "auth0|123456",
  "aud": "1234abcdef",
  "exp": 1311281970,
  "iat": 1311280970,
  "name": "Jane Doe",
  "given_name": "Jane",
  "family_name": "Doe"
}
```

Apart from the technical data about the server that issued the token, the expiration date, etc., here you can find three claims about the user: `name`, `given_name`, and `family_name`.

In its basic operation, we can represent the OpenID Connect flow as follows:

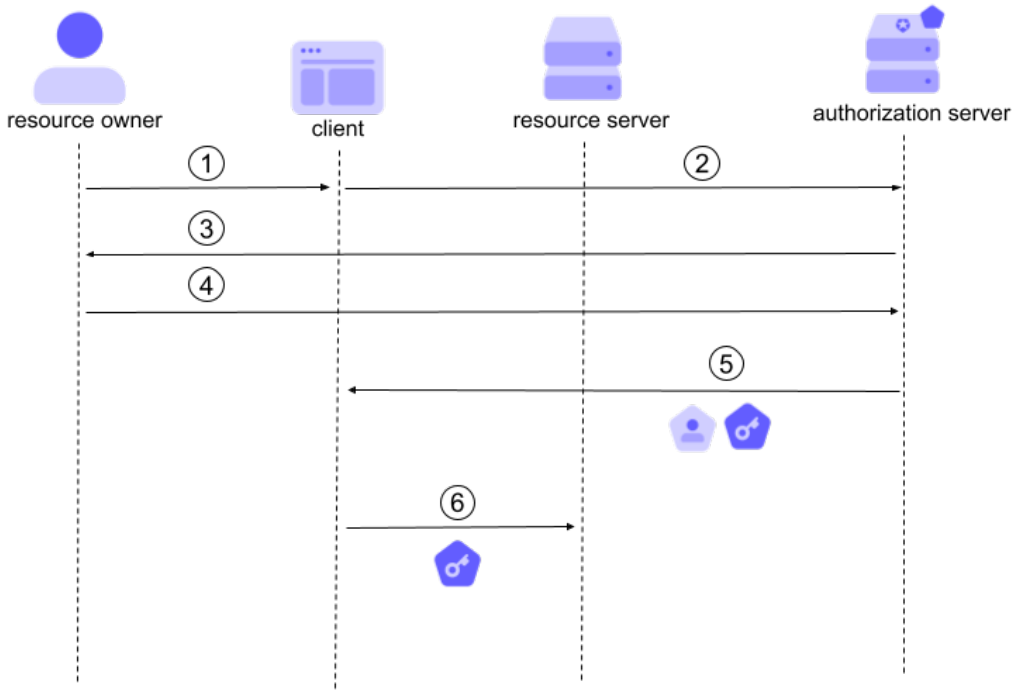


This flow aims only to authenticate the user. No access to a resource is included. Here are the steps described:

1. The user accesses the client application.
2. The client redirects the user to the authorization server for authentication,
3. The authorization server asks the user for their credentials. Typically, a login form is shown in this step, but other authentication factors can be requested.
4. The user provides their credentials to the authorization server.
5. After successful authentication, the authorization server sends an ID token to the client application.

Since OpenID Connect is based on OAuth 2.0, it is integrated into its authorization flows. This means that if the client needs to access a resource, this request is made simultaneously with user authentication.

Let's see the generic flow in this case:



auth0.com | © Auth0 2022

In this case, the client application also requests access to the user's resource. If the user gives their consent, the client application will receive an access token in addition to the ID token in step 5. Then, it will use the access token to access that resource (step 6).

> Remember: the ID token is the result of the authentication process and may contain data about the user profile; the access token authorizes your application to access a resource on your behalf.
>
> Don't use the ID token to access a user's resource.

Summary

This chapter quickly summarized some of the basic concepts about modern identity and access management used by Auth0. You learned about authentication and authorization and received a high-level overview of some of the most common access control models: permissions-based, RBAC, ABAC, and ReBAC.

You also received an overview of OpenID Connect and OAuth, the two standard authentication and authorization protocols used by most implementers of the so-called modern identity, including Auth0.

auth0.com | © Auth0 2022

Chapter 2 - Introduction to Auth0

auth0.com | © Auth0 2022

In the previous chapter, you learned about authentication and authorization. You got a sneak peek at different authentication factors, authorization models, and modern identity standards. You may have realized that there is a lot of complexity behind that simple login box that allows users to access your application. However, while having a high-level understanding of that complexity can help, you can ignore the implementation details and focus only on the business logic of your application: Auth0 can help you with this.

In this chapter, you'll take a quick look at the Auth0 platform's main features that help you simplify authentication and authorization implementation in your applications. It will focus on how the interaction between your application and the Auth0 platform works and the tools allow you to control this interaction.

What is Auth0?

Auth0 is an **Identity and Access Management** (**IAM**) platform that simplifies the burden of authenticating and managing your application's users. It provides authentication and authorization as a service so that you can focus on building the core features of your application. Beyond the classic login form, Auth0 offers security out of the box and a number of features to improve the user's authentication experience and customize the various flows following your needs.

Using Auth0, developers significantly reduce the cost of building an authentication and authorization system and avoid the risks of keeping their homemade solution secure.

Auth0 allows users to log in to your application in several ways, based on your configuration:

- Through the classic username and password.
- Using multi-factor authentication (MFA).

auth0.com | © Auth0 2022

- Via social accounts like Google, Twitter, LinkedIn, etc.
- With passwordless login.
- Using biometrics.

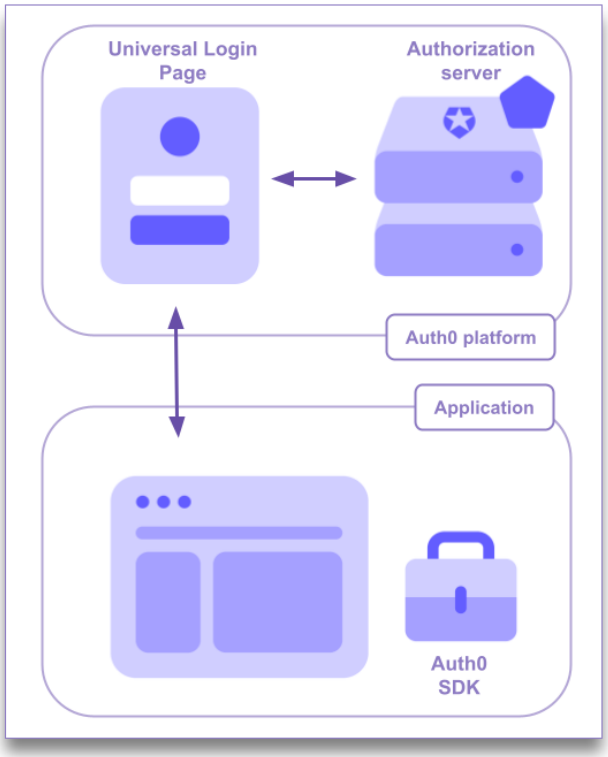
It supports **Single Sign-On** (**SSO**), and its services are based on industry standards such as **OAuth2**, **Open ID Connect** (**OIDC**), and **SAML**.

You can use Auth0 services independently of the application type and the specific programming language or development framework.

Your Application and Auth0

You will learn in practice how to integrate your .NET applications with Auth0 in the next chapters. However, for now, it is worth gaining a high-level overview of the relationship between an application and the Auth0 platform.

Let's take a look at the following picture:



auth0.com | © Auth0 2022

This diagram illustrates the main components involved in the authentication process with Auth0. You can see two actors:

- The application
- The Auth0 platform

The application uses the Auth0 SDK, a library specific to the programming language or framework your application is built with. The Auth0 SDK simplifies the interaction between your application and the Auth0 platform hiding the underlying complexity. Auth0 provides **plenty of SDKs** for different development environments. In some cases, you may not need a specific SDK to interact with the Auth0 platform since the framework you are using has native support for OAuth or OIDC. You will learn more about each specific case in the following chapters.

The relevant Auth0 components highlighted in the diagram are:

- The **Auth0 Universal Login page**, which is the web page that allows the user to authenticate.
- The **Auth0 authorization server**, which is the server that handles authentication and authorization requests.

Briefly, when users access a protected area of your application, they are redirected to the Universal Login page to authenticate with Auth0. After users are successfully authenticated, they are redirected back to your application with the credentials that allow them to access the protected area.

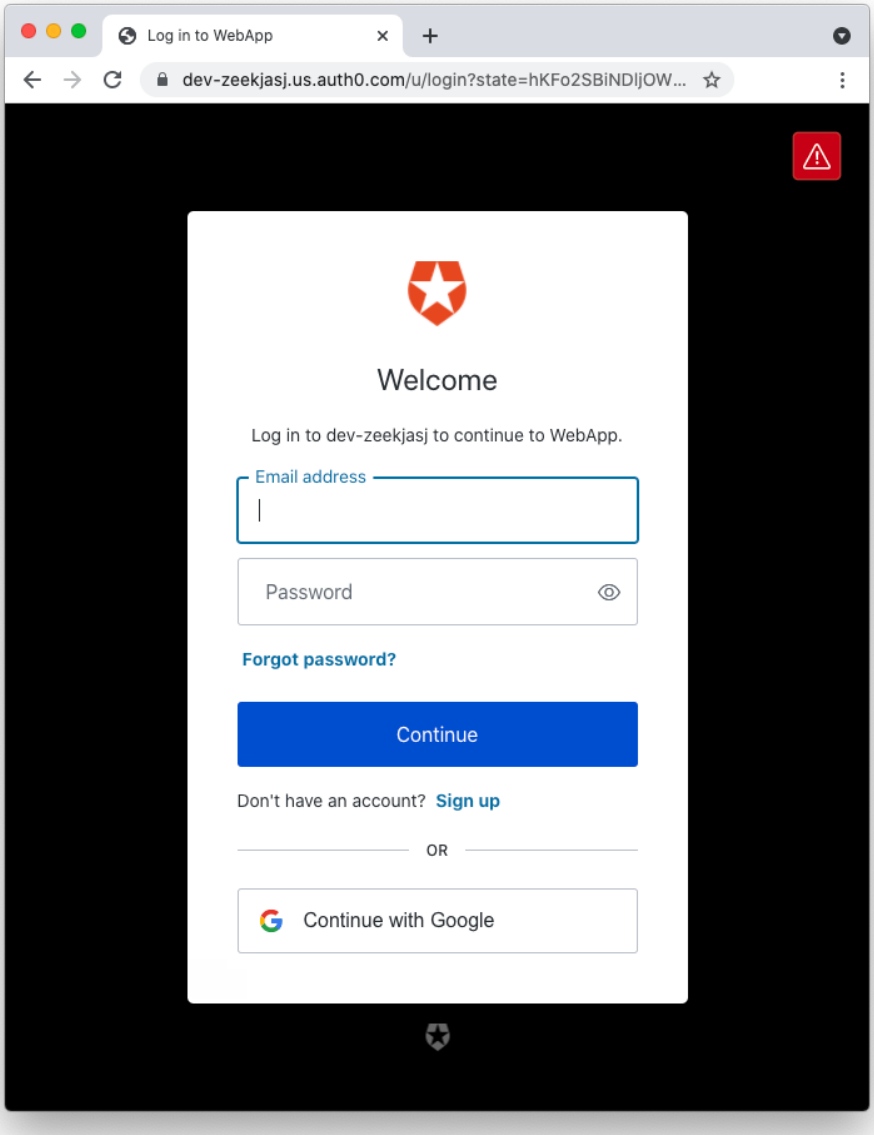
The Auth0 Universal Login Page

When users authenticate with Auth0, they will be redirected to a web page showing the typical login box request. This page is called the **Auth0 Universal Login** page. It will be automatically opened whenever an

auth0.com | © Auth0 2022

authentication or authorization request is triggered by your application and your user is not authenticated.

In its basic form, the Auth0 Universal Login page appears as in the following picture:

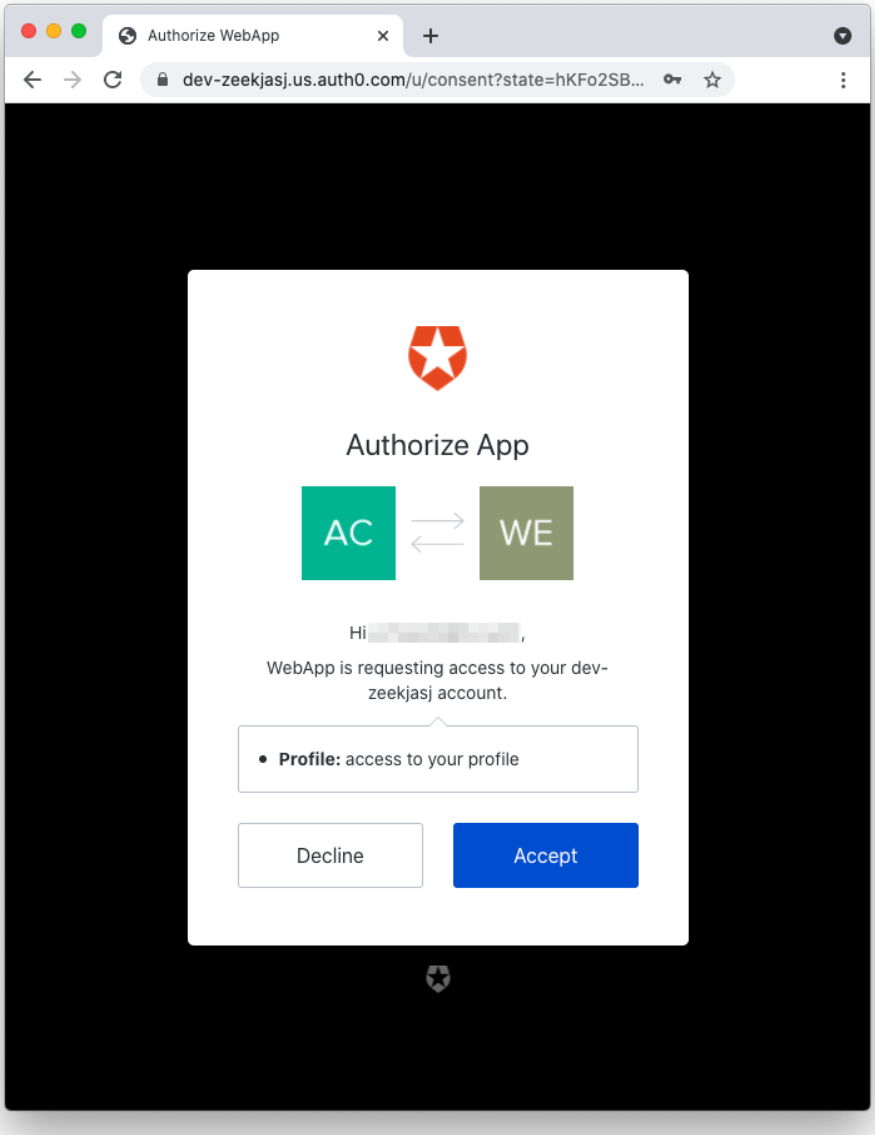


As you can see, it is a classic form request for a username and password. It also includes a link to recover a forgotten password and a link to sign up as a new user. In addition, it allows you to log in using your Google account.

auth0.com | © Auth0 2022

All these features are available out of the box. You don't need to configure anything specific. But that is not all. You can **customize the login page** to fit your needs and be consistent with your brand. Also, you can remove Google account integration, add other social identity providers, or connect with other identity sources.

Usually, the first time your users authenticate with Auth0 through your application, they will see a page like the following:



This is known as the **consent screen**. This page informs the user that your application may access their personal data and asks for their explicit authorization. This screen is shown only at the very first time any of your users access your application.

The Auth0 Dashboard

By integrating Auth0 into your application, you actually delegate authentication and authorization functionality to the Auth0 platform. This means that your application trusts Auth0 and lets it authenticate your users. The user authentication process happens on the Auth0 side instead of in your application. For this reason, your application needs to know how to redirect users to Auth0 when they need to authenticate. On the other side, Auth0 needs to understand how to redirect users back to your application once they authenticate.

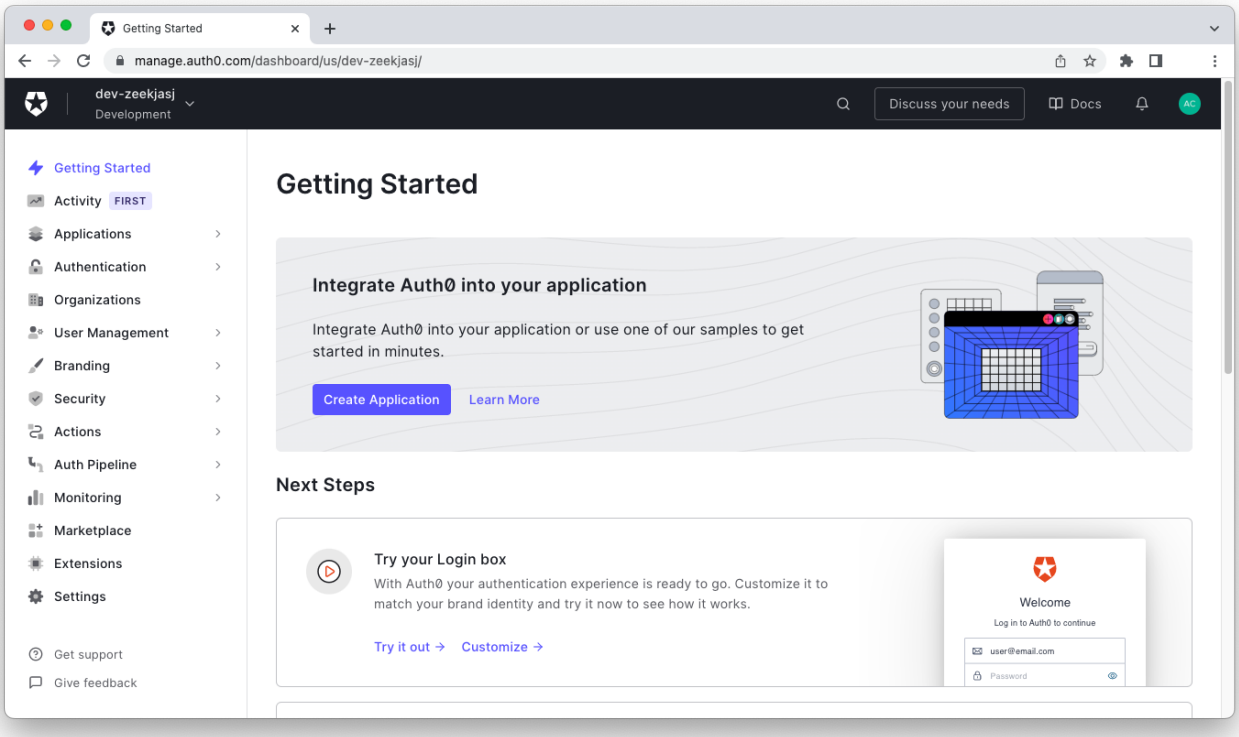
You need to register your application to let it interact with the Auth0 platform and vice versa. Indeed the registration process has a twofold purpose:

- To let Auth0 know about your application
- To get some data that allows your application to know about Auth0

To register your application with Auth0, you need an account. If you don't already have one, you can **sign up** for a free Auth0 account.

Your Auth0 account allows you to access the **Auth0 dashboard**, which is your main reference point for all your Auth0 settings. Here is the home page of the Auth0 dashboard:

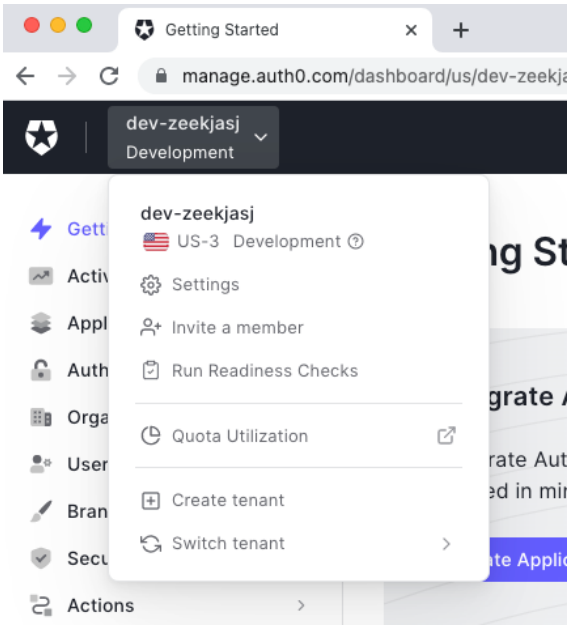
auth0.com | © Auth0 2022



An important concept you should become familiar with is the Auth0 tenant.

> An **Auth0 tenant** is an isolated collection of configuration assets that defines how you use Auth0 authentication and authorization services.

When you sign up with Auth0, a tenant is created for you. You can see its name and access its settings at the top left corner of your Auth0 dashboard:



Your tenant is identified by a DNS domain name. By default, this is a third-level domain of the `auth0.com` domain. It is in the format `YOUR-TENANT-NAME.auth0.com`, where `YOUR-TENANT-NAME` is the name assigned to your tenant. The domain name associated with your tenant is also known as the **Auth0 domain**.

You can also use your own second-level domain name to identify your tenant instead of `auth0.com`. Check out the documentation about **custom domains** to learn more.

You can create multiple tenants, typically based on your deployment environment. So you will have a development and a production tenant as well as a stage tenant.

To learn more about tenants, read **the documentation**.

The most visible role of your Auth0 dashboard may be the ability to let you register your application with Auth0 and get the Auth0 settings you need to configure your application. Be aware that the Auth0 dashboard allows you to do many more things, such as customizing the Universal Login page, extending the user registration and login processes with Actions, and much more.

> As an alternative to the Auth0 dashboard, you can use **the Auth0 CLI** to configure Auth0. In this case, **install the CLI on your machine** and **configure it to access your Auth0 tenant**.

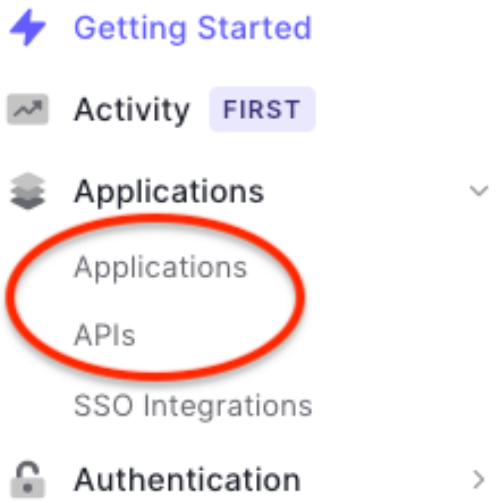
Application Types

We mentioned that the Auth0 platform works with any application type. While this is true, you should pay attention to the type of your application because different flows apply for authentication and authorization, as you learned reading about OpenID Connect and OAuth2.

auth0.com | © Auth0 2022

APIs and applications

The first primary distinction is between proper applications and APIs. You can find this distinction at the menu level:

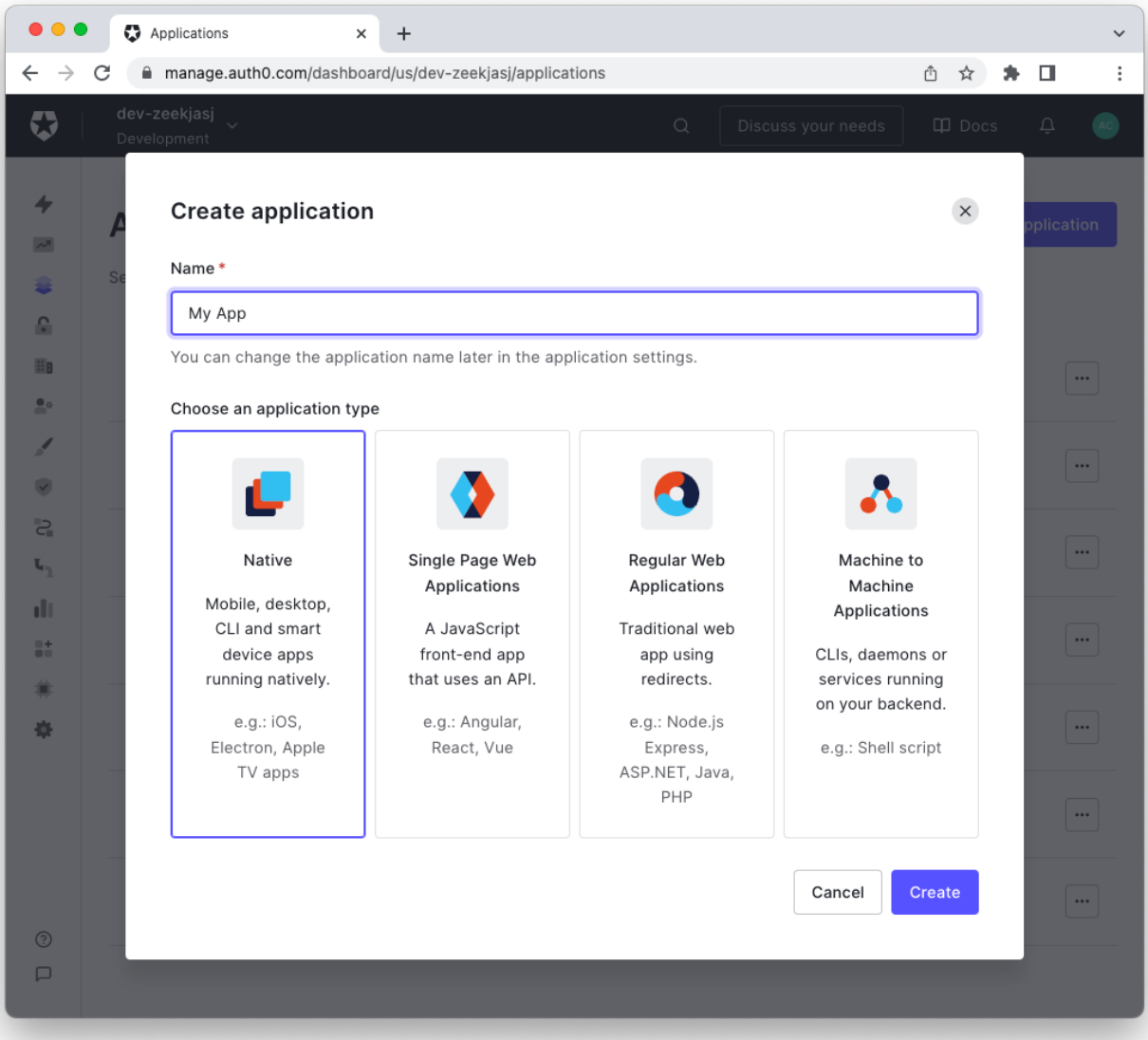


This distinction is related to the different roles that an API and other types of applications have within the OpenID Connect and OAuth2 flows. Do you remember the OAuth actors? In this context, an application is an OAuth client, while an API is a resource server.

Register your application

To register an application through the Auth0 dashboard, go to the **Applications section** and click the *Create Application* button. You'll get the following screen:

auth0.com | © Auth0 2022



The dialog requests you insert a friendly name for your application and select the application type. You can choose from the following types according to your application:

- **Native**. This category includes desktop and mobile applications as well as smart device apps such as smart TV apps. Typical .NET applications in this category are Windows Forms and WPF applications, Xamarin Forms applications, and .NET MAUI applications.

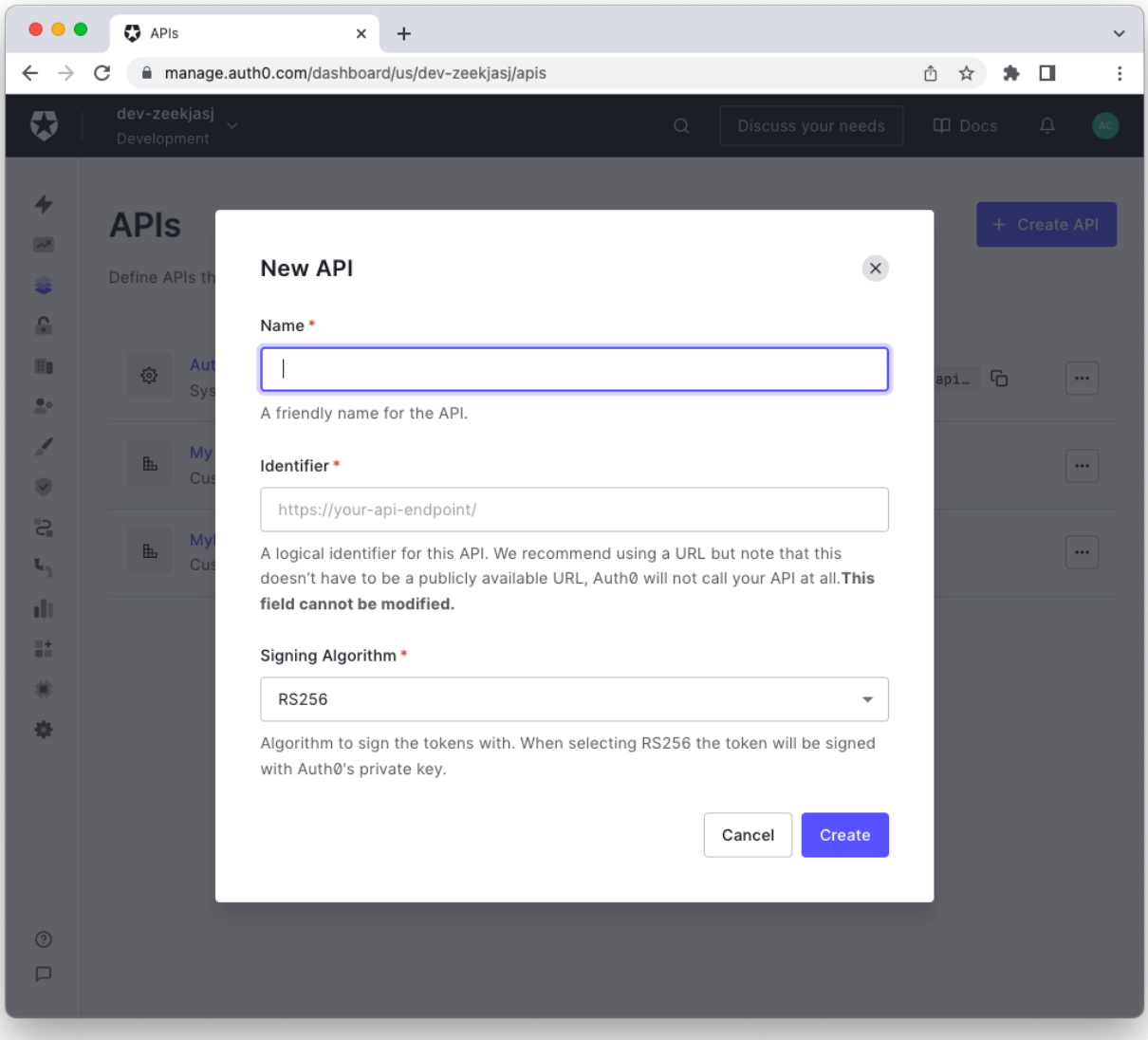
© Auth0 2022 | auth0.com

- **Single Page Applications** (**SPA**). These are JavaScript applications that perform most of their user interface logic in a web browser and call APIs for backend services. They are usually built with frameworks such as Angular, React, Vue, etc. In the .NET context, Blazor WebAssembly applications fall in this category.
- **Regular Web Applications**. These are the traditional web applications that perform most of their business logic on the server, including most of the UI rendering. ASP.NET Core MVC apps, Razor Pages apps, and Blazor Server pages are examples of .NET regular web applications.
- **Machine-to-Machine Applications** (**M2M**). You can consider any non-interactive application a member of this category, such as tasks, daemons, and services running on your server and IoT devices. JavaScript, into the vulnerable website.

As you may guess, the reason for this further application classification is related to the different OAuth flows you have to implement depending on your application type.

Register your API

To register your API with Auth0, go to the **API section** of your Auth0 dashboard and click the *Create API* button. The following dialog opens:



In this case, you have to provide a friendly name and an **API identifier**, also known as **audience**. The latter is a unique identifier in the URI format, such as `http://my-great-api`. Although this identifier is in the URI format, it doesn't need to be a reachable URL. Auth0 will not call it but will use it to identify your API.

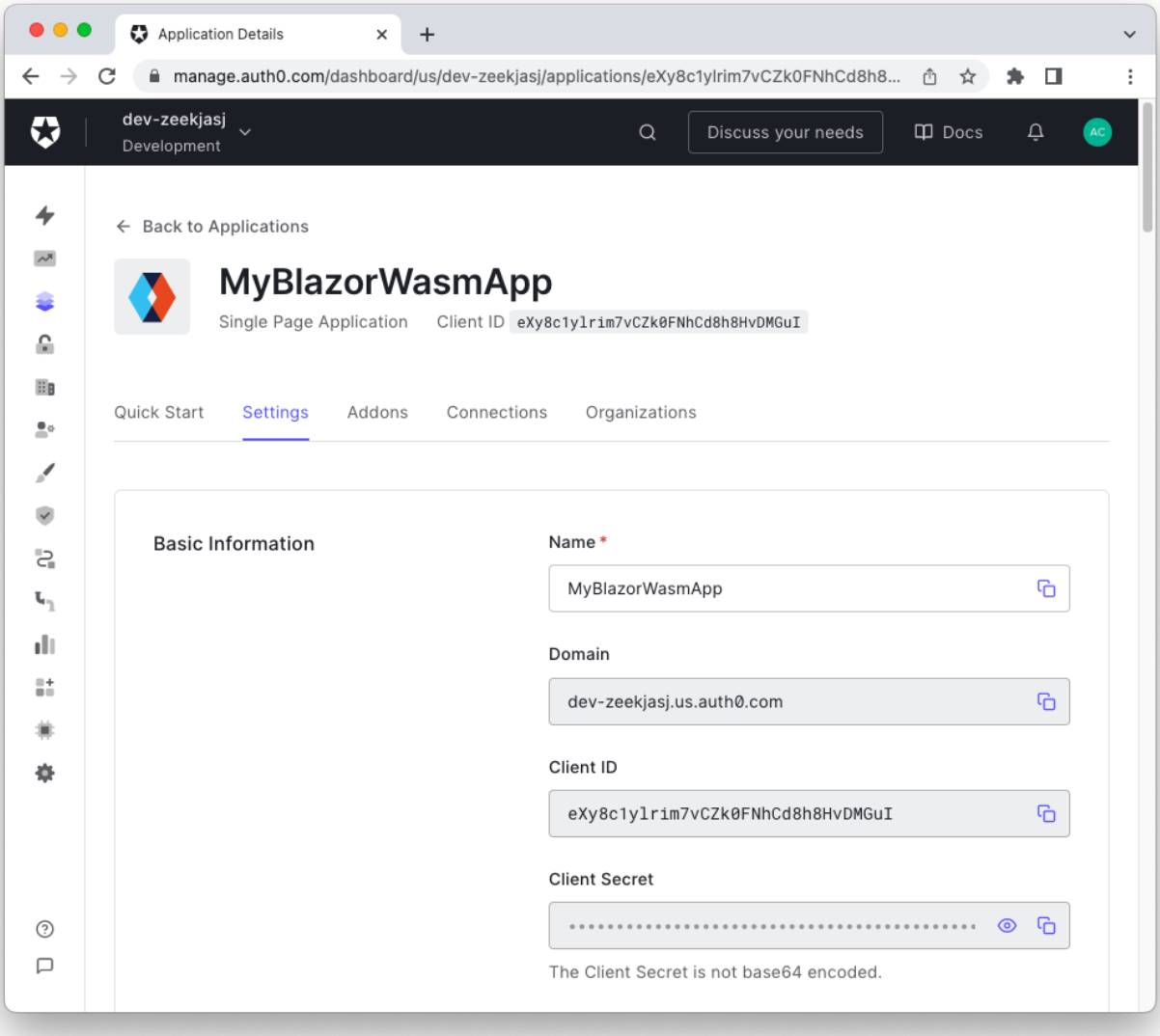
Keeping the default RS256 signing algorithm is highly recommended.

auth0.com | © Auth0 2022

Application Settings

Once you create your application or API on the dashboard, Auth0 will generate some settings for you. You can find them in the *Settings* tab of your application or API. In the same tab, you can insert or modify some values to adjust the Auth0 behavior to your application's needs.

The following picture shows the basic information in the *Settings* tab of an application:

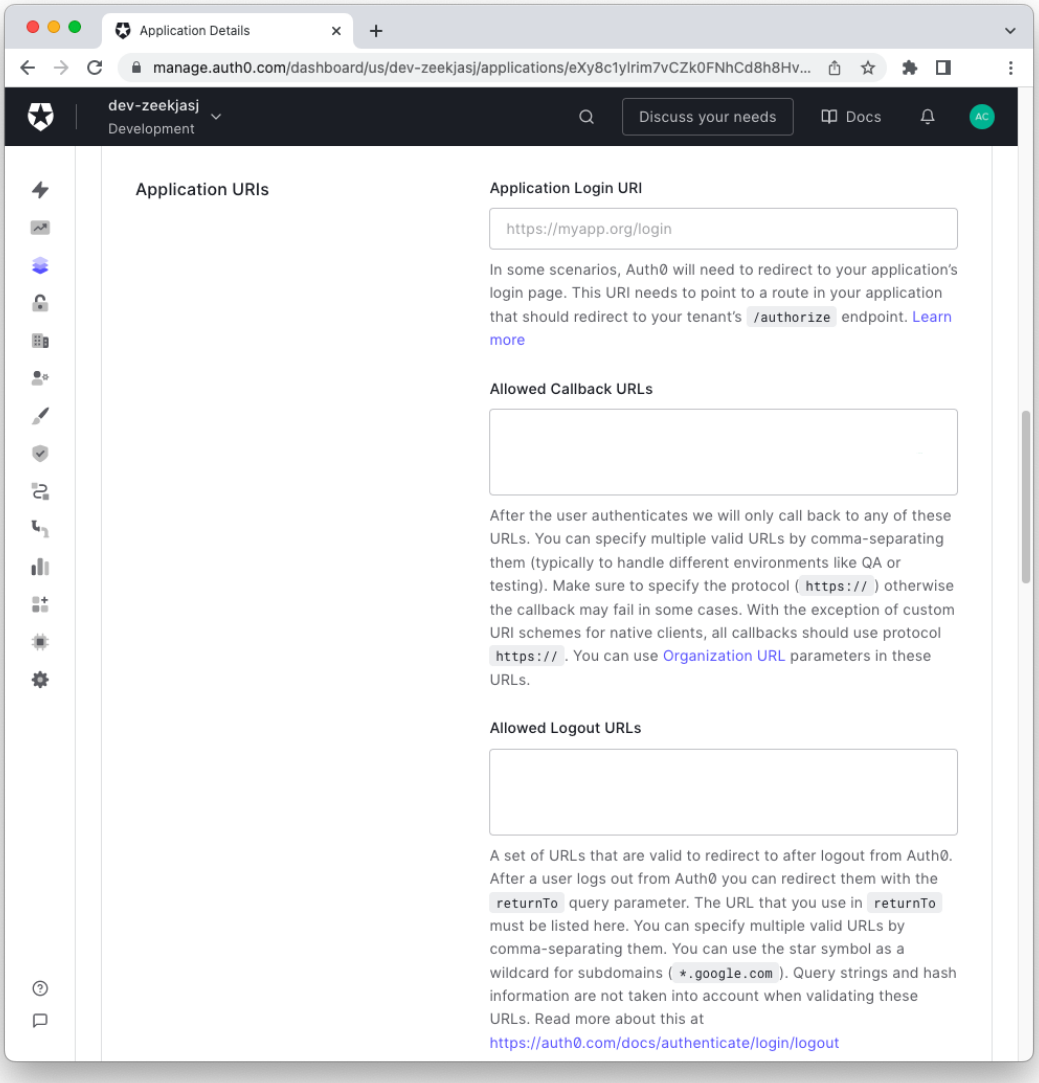


© Auth0 2022 | auth0.com

The main settings generated by Auth0 for your application are:

- The **Auth0 domain**. This is the domain name automatically assigned to your tenant. It allows your application to reach out to Auth0 to authenticate users.
- The **client ID**. It's a unique identifier for your application. It allows Auth0 to identify your application and can be considered public information.
- The **client secret**. This is a string to be used as a password for your application with some OAuth2 flows. This is confidential information and should be protected from unauthorized access.

In the same tab, you will find the *Application URIs* section, as shown below:



auth0.com | © Auth0 2022

This section lets you specify some URIs of your applications. These URIs allow Auth0 to know how to interact with your application. Two fields are particularly relevant:

- **Allowed Callback URLs**. This field allows you to specify one or more URLs where Auth0 can redirect users after they authenticate.
- **Allowed Logout URLs**. In this field, you can specify one or more URLs where Auth0 can redirect users after they log out of your application.

Registration of these URLs is required for security reasons. It prevents an attacker from injecting an unauthorized URL in your authorization request and stealing the tokens issued for your application.

> Don't worry if not everything is clear to you at the moment. You'll learn more about these settings and how to use them in the next chapters.

Summary

In this chapter, you became acquainted with the main features of Auth0 that let you integrate your application with its authentication and authorization services. After a high-level overview of the interaction between your application and the Auth0 platform, you met the Auth0 Universal Login page and the consent screen.

Then, you were introduced to the Auth0 dashboard and learned about the tenant concept, under which you register your application and get the needed settings to configure its integration.

auth0.com | © Auth0 2022

Chapter 3 - Regular Web Applications

auth0.com | © Auth0 2022

The .NET platform provides developers with the ASP.NET Core framework to create web applications. ASP.NET Core provides developers with a rich set of functionalities to build applications that use the Web as their target platform.

This chapter will focus on **regular web applications**, also known as **traditional web applications**. These are applications that perform most of their business logic on the server, including most of the UI rendering.

The ASP.NET Core framework allows you to use different programming models: **ASP.NET Core MVC**, **Razor Pages**, and **Blazor Server**. Actually, the difference between these three models is not so clear-cut. ASP.NET Core MVC applications use Razor Pages and the same happens for Blazor Server applications. On the other side, Blazor Server applications can implement controllers and views following the MVC design pattern.

This chapter will address how to integrate Auth0 into applications that use each of the three ASP.NET Core programming models for building regular web applications.

auth0.com | © Auth0 2022

ASP.NET Core MVC

ASP.NET Core MVC is probably the most popular framework for building web applications in .NET. It uses the well-known **Model-View-Controller design pattern**, which helps you achieve separation of concerns for your application UI. In this section, you will learn how to integrate authentication in an ASP.NET Core MVC application by using the Auth0 ASP.NET Core Authentication SDK.

The sample application

To show how to integrate an ASP.NET Core MVC application with Auth0, you'll create a simple app based on the standard .NET template. In a terminal window, run the following command:

```
dotnet new mvc -o Auth0Mvc
```

After the command creates the ASP.NET Core MVC application, go to the `Auth0Mvc` folder and launch the application with the following command:

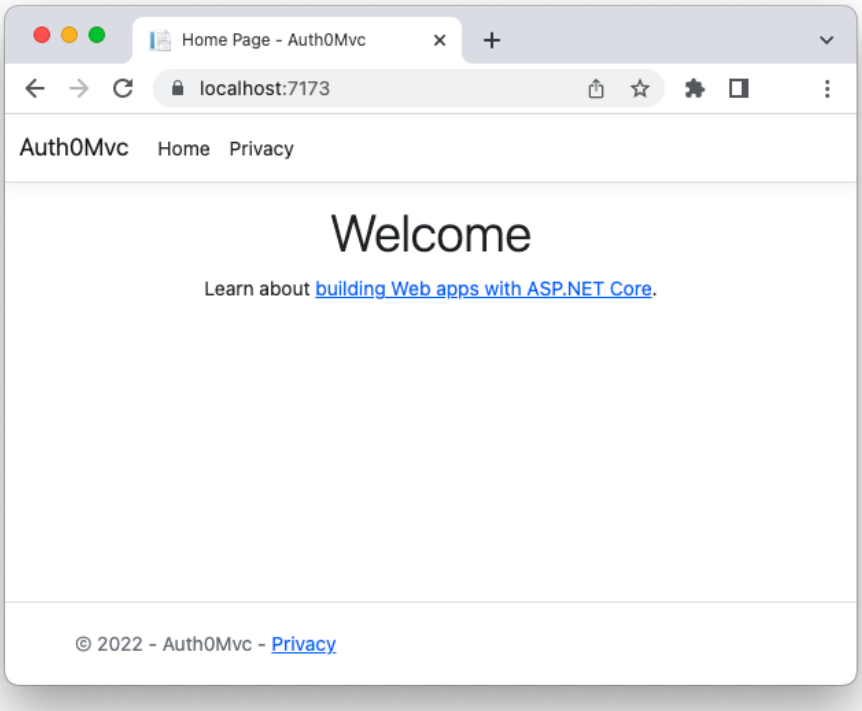
```
dotnet run
```

> If you are using a Mac, you may be affected by a known issue when running an ASP.NET Core application through the .NET CLI. Please refer to **this note** to learn about a workaround.

From your terminal, take note of the address your application is listening to and navigate to it with your browser. In the example shown in this section, the application's address is `https://localhost:7173`. Replace this address with yours wherever it's needed.

auth0.com | © Auth0 2022

If everything works as expected, you should see the following page:



You are now ready to integrate your application with Auth0.

Register with Auth0

The first step is to register the application with Auth0. **Chapter 2** already introduced you to registering the app. Let's quickly recall the process here. In your dashboard, go to the **Applications section** and follow these steps: Let's focus on the relevant point of this investigation:

1. Click on *Create Application*.
2. Provide a friendly name for your application (for example, *ASP.NET Core Web App*) and select *Regular Web Applications* as the application type.
3. Finally, click the *Create* button.

These steps make Auth0 aware of your ASP.NET Core MVC application.

After the application has been created, go to the Settings tab and take note of your **Auth0 domain** and your **client ID**. Then, in the same form, assign the value `https://localhost:<YOUR_PORT_NUMBER>/callback` to the *Allowed Callback URLs* field and the value `https://localhost:<YOUR_PORT_NUMBER>/` to the *Allowed Logout URLs* field. Replace the `<YOUR_PORT_NUMBER>` placeholder with the actual port number assigned to your application.

The first value tells Auth0 which URL to call back after the user authentication. The second value tells Auth0 which URL a user should be redirected to after their logout.

Click the *Save Changes* button to apply your changes.

Now, go back to the root folder of the sample application project, open the `appsettings.json` configuration file, and replace its content with the following:

```
// appsettings.json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Auth0": {
    "Domain": "YOUR_DOMAIN",
    "ClientId": "YOUR_CLIENT_ID"
  }
}
```

© Auth0 2022 | auth0.com

Replace the placeholders `YOUR_DOMAIN` and `YOUR_CLIENT_ID` with the respective values taken from the Auth0 dashboard.

Add authentication

At this point, the basic settings for connecting your application to Auth0 are in place. To add authentication, you need to apply some changes to your application and use those settings. Let's go one step at a time.

Install the Auth0 SDK

As your first step, install the **Auth0 ASP.NET Core Authentication SDK** by running the following command in your terminal window:

```
dotnet add package Auth0.AspNetCore.Authentication
```

The Auth0 ASP.NET Core SDK lets you easily integrate OpenID Connect-based authentication in your app without dealing with all its low-level details.

If you want to learn a bit more, **this article** provides you with an overview of the Auth0 ASP.NET Core SDK.

Set up authentication

Now, let's modify the application code to support authentication. Open the `Program.cs` file and change its content as follows:

© Auth0 2022 | auth0.com

```
// Program.cs

using Auth0.AspNetCore.Authentication; // 👈 new code

var builder = WebApplication.CreateBuilder(args);

//  👇 new code
builder.Services
    .AddAuth0WebAppAuthentication(options => {
      options.Domain = builder.Configuration["Auth0:Domain"];
      options.ClientId = builder.Configuration["Auth0:ClientId"];
    });
// ☝️ new code

builder.Services.AddControllersWithViews();

// ...existing code...

app.UseRouting();

app.UseAuthentication(); // 👈 new code
app.UseAuthorization();

// ...existing code...
```

Following the highlighted code, you added a reference to the `Auth0.AspNetCore.Authentication` namespace at the beginning of the file. Then you invoked the `AddAuth0WebAppAuthentication()` method with the Auth0 domain and client id as arguments. Finally, you called the `UseAuthentication()` method to enable the authentication middleware. Make sure to call `UseAuthentication()` **before** `UseAuthorization()`.

These changes lay the groundwork for supporting authentication via Auth0.

Implement login

To implement login, add a `AccountController.cs` file to the `Controllers` folder with the following content:

```
// Controllers/AccountController.cs

using Microsoft.AspNetCore.Authentication;
using Auth0.AspNetCore.Authentication;
using Microsoft.AspNetCore.Mvc;

public class AccountController : Controller
{
    public async Task Login(string returnUrl = "/")
    {
        var authenticationProperties = new LoginAuthenticationPropertiesBuilder()
            .WithRedirectUri(returnUrl)
            .Build();

        await HttpContext.ChallengeAsync(Auth0Constants.AuthenticationScheme,
        authenticationProperties);
    }
}
```

This class defines a controller meant to handle typical user account operations. In the code above, you implemented the login operation through the `Login()` method. This method creates a set of authentication properties required for the login and triggers the authentication process via Auth0.

To enable this login operation, you need to change the UI. So, open the `_Layout.cshtml` file under the `Views/Shared` folder and update its content as follows:

auth0.com | © Auth0 2022

```
// Views/Shared/_Layout.cshtml

<!DOCTYPE html>
<html lang="en">

<!-- ...existing code -->

  <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
    // 👇 new code
    @if (User.Identity.IsAuthenticated)
    {
    // 👆 new code
    <ul class="navbar-nav flex-grow-1">
      <li class="nav-item">
        <a class="nav-link text-dark"
           asp-area=""
           asp-controller="Home"
           asp-action="Index">Home</a>
      </li>
      <li class="nav-item">
        <a class="nav-link text-dark"
           asp-area=""
           asp-controller="Home"
           asp-action="Privacy">Privacy</a>
      </li>
    </ul>
    // 👇 new code
    } else {
      <ul class="navbar-nav ms-auto">
        <li class="nav-item">
          <a class="nav-link text-dark"
             asp-area=""
```

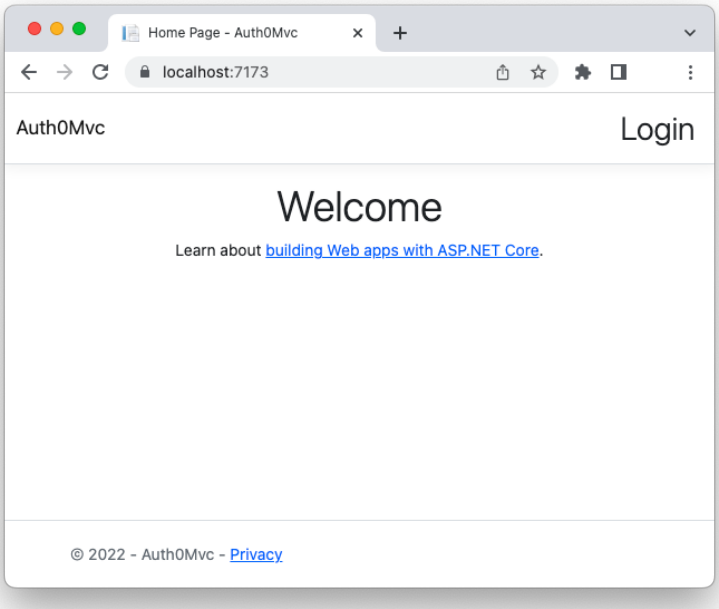
auth0.com | © Auth0 2022

```
                asp-controller="Account"
                asp-action="Login">Login</a>
            </li>
          </ul>
          }
          // 👆 new code
      </div>

<!-- ...existing code -->
```

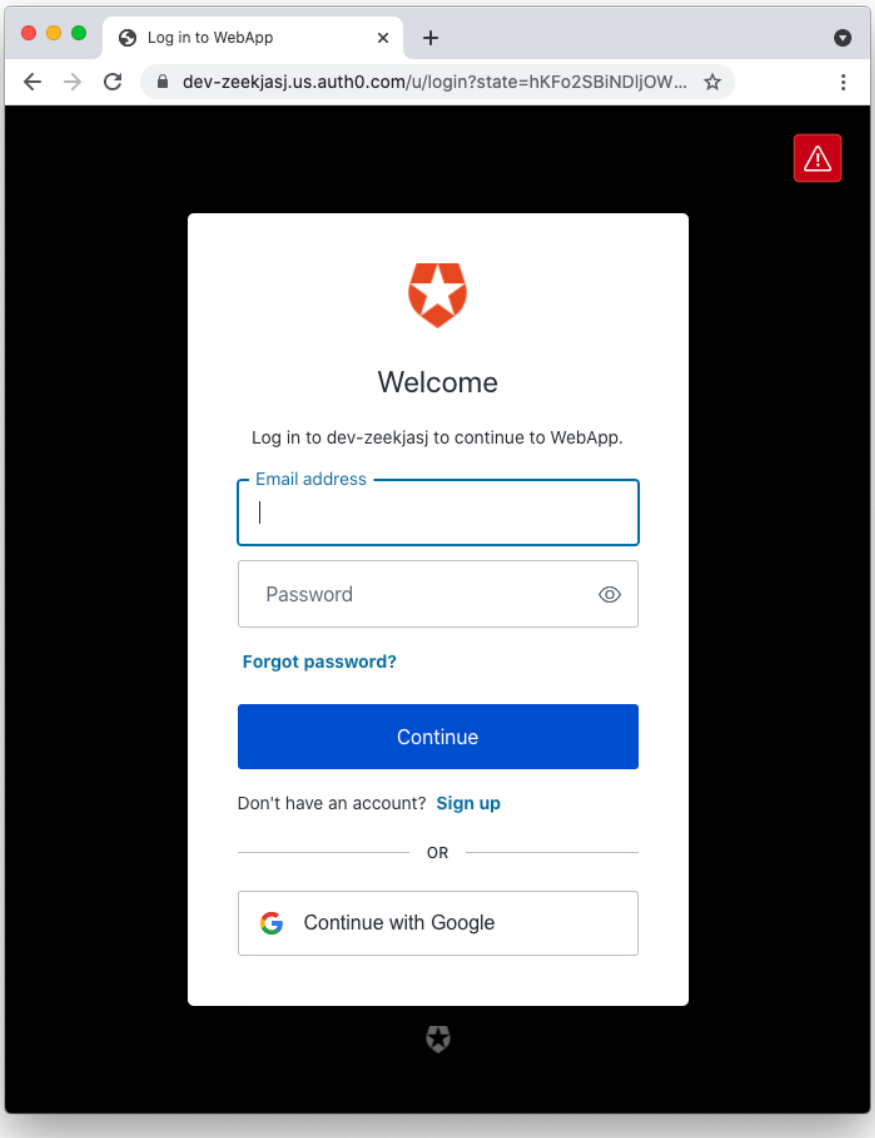
You modify the page's markup by adding a check on the `User.Identity.IsAuthenticated` property. As you may have figured out, this property lets you know whether the current user is authenticated or not. If the user is not authenticated, a *Login* link will be shown on the right side of the navigation bar.

Test login

Everything is ready. Go to your browser, and you should see the *Login* link as shown in the following picture:

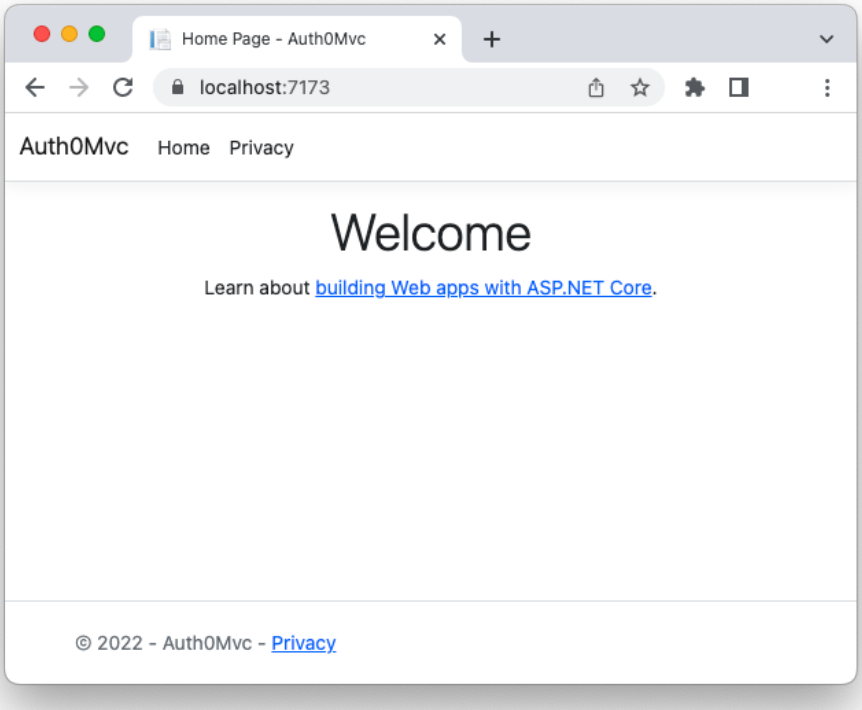


Notice that, this time, you can't see the *Home* and *Privacy* links on the left side of the home page. When you click the *Login* link, you will be redirected to the **Auth0 Universal Login page**, as shown in the following screenshot:



You have already become acquainted with this page in **Chapter 2**. After entering the user's credentials and accepting the consent screen, you are redirected back to the home page, which will be shown as before:

© Auth0 2022 | auth0.com



Congratulations! You added authentication to your ASP.NET Core MVC application!

Protect private views

Although you have to authenticate to see the *Home* and *Privacy* links in the navigation bar, the Privacy view itself is not protected. You can verify this by accessing the `https://localhost:7173/Home/Privacy` address directly or by clicking the link in the page's footer.

If you want the Privacy page to be accessible only to authenticated users, you need to restrict access to it.

To do this, edit the `HomeController.cs` file under the `Controllers` folder as shown below:

auth0.com | © Auth0 2022

```
// Controllers/HomeController.cs

using System.Diagnostics;
using Microsoft.AspNetCore.Mvc;
using Auth0Mvc.Models;
using Microsoft.AspNetCore.Authorization; // 👈 new code

namespace Auth0Mvc.Controllers;

public class HomeController : Controller
{

  // ...existing code...

  [Authorize] // 👈 new code
  public IActionResult Privacy()
  {
    return View();
  }

  // ...existing code...

}
```

You added a reference to the `Microsoft.AspNetCore.Authorization` namespace, which provides you with the `Authorize` attribute. You use this attribute by attaching it to the `Privacy()` method of the controller. This method renders the `Privacy.cshtml` view defined in the `Views/Home` folder. This simple change enables the authorization check on the *Privacy* view so that only authenticated users can access it.

Now, if you try to access the *Privacy* page before authenticating, you are redirected to the Auth0 Universal Login page.

auth0.com | © Auth0 2022

Implement Logout

Another missing item needed to make your application usable is logout. In fact, currently, when you log in to the application, you stay logged in until your session on Auth0 expires. You may want to allow your user to explicitly log out of the application. For this purpose, open the `AccountController.cs` file in the `Controllers` folder and add the code highlighted below:

```
// Controllers/HomeController.cs

using Microsoft.AspNetCore.Authentication;
using Auth0.AspNetCore.Authentication;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization; // 👈 new code
using Microsoft.AspNetCore.Authentication.Cookies; // 👈 new code

public class AccountController : Controller
{

  // ...existing code...

  // 👇 new code
  [Authorize]
  public async Task Logout()
  {
    var authenticationProperties = new LogoutAuthenticationPropertiesBuilder()
        .WithRedirectUri(Url.Action("Index", "Home"))
        .Build();

    await HttpContext.SignOutAsync(Auth0Constants.AuthenticationScheme,
    authenticationProperties);
```

© Auth0 2022 | auth0.com

```
    await HttpContext.SignOutAsync(CookieAuthenticationDefaults.Authentication
    Scheme);
  }
  // 👆 new code
}
```

You added the references to the `Microsoft.AspNetCore.Authorization` and `Microsoft.AspNetCore.Authentication.Cookies` namespaces and defined a new method for the `AccountController` class: `Logout()`. This method creates a set of required properties and triggers the logout process to destroy both the Auth0 session and the local session.

The next step is to make the logout link available to the user. So, update the content of the `_Layout.cshtml` file under the `Views/Shared` folder as follows:

```
// Views/Shared/_Layout.cshtml

<!DOCTYPE html>
<html lang="en">

<!-- ...existing code -->

  <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
    @if (User.Identity.IsAuthenticated)
    {
      <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
          <a class="nav-link text-dark"
             asp-area=""
             asp-controller="Home"
             asp-action="Index">Home</a>
```

© Auth0 2022 | auth0.com

```
        </li>
        <li class="nav-item">
          <a class="nav-link text-dark"
             asp-area=""
             asp-controller="Home"
             asp-action="Privacy">Privacy</a>
        </li>
      </ul>
      // 👇 new code
      <ul class="navbar-nav ms-auto">
        <li class="nav-item">
          <a class="nav-link text-dark"
             asp-area=""
             asp-controller="Account"
             asp-action="Logout">Logout</a>
         </li>
       </ul>
      // 👆 new code
    } else {
      <ul class="navbar-nav ms-auto">
        <li class="nav-item">
          <a class="nav-link text-dark"
             asp-area=""
             asp-controller="Account"
             asp-action="Login">Login</a>
        </li>
      </ul>
      }
  </div>

<!-- ...existing code -->
```

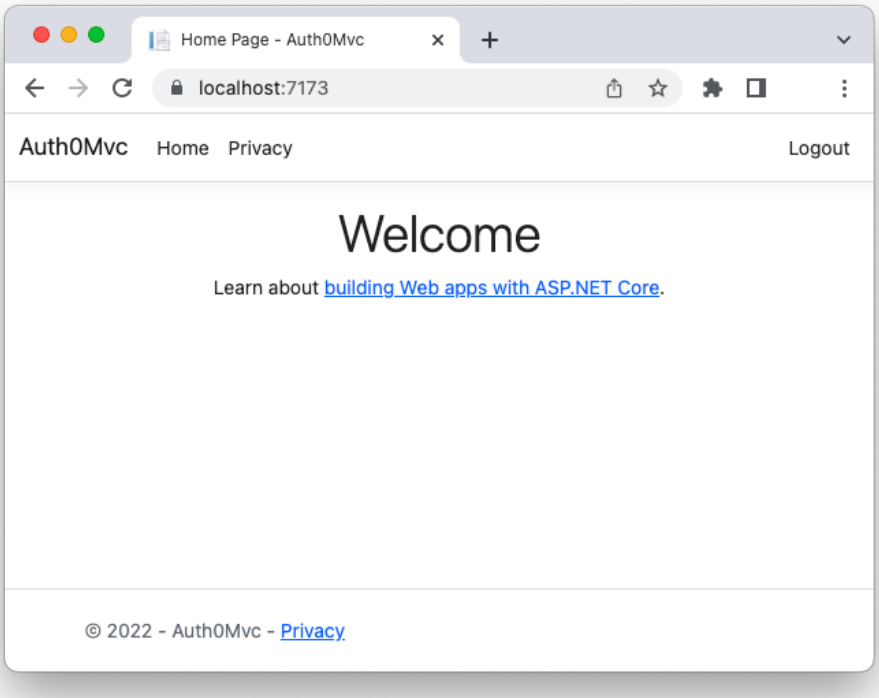
© Auth0 2022 | auth0.com

You just added the logout link to the right side of the navigation bar when the user is authenticated. That link points to the `Logout()` method of the `AccountController` class.

Test logout

It's time to test this new version of the application. First, **delete all cookies from your browser or use an incognito window**. This is needed because you haven't had the chance to log out so far.

Now, after you log in again, the home page should look as follows:



When you click the *Logout* link, you will be disconnected from Auth0 and see the usual home page with only the *Login* link.

Add a user profile page

Let's try to go one step further and add a page showing some data about the user.

Specify your scopes

As said before, the Auth0 ASP.NET Core Authentication SDK uses OpenID Connect (OIDC) to authenticate your users. OIDC provides your application with an **ID token** containing some basic data about the user. From a technical point of view, this user data is available because the SDK requests the `openid` and `profile` **scopes** by default. You don't see this because the SDK takes care of the whole authentication process and manages the ID token. So, by default, you have the user's name and possibly their picture. If you also want the user's email in their profile, you have to specify the scopes explicitly.

Open the `Program.cs` file and apply the following change:

```
// Program.cs

// ...existing code...

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddAuth0WebAppAuthentication(options => {
      options.Domain = builder.Configuration["Auth0:Domain"];
      options.ClientId = builder.Configuration["Auth0:ClientId"];
      options.Scope = "openid profile email"; // 👈 new code
    });

// ...existing code...
```

auth0.com | © Auth0 2022

You assigned a string to the `Scope` option containing the default scopes mentioned before (`openid` and `profile`) and the `email` scope.

This causes the email to be added to the user profile.

Create the profile model

To show the user profile, let's start by creating a model. Add a `UserProfileViewModel.cs` file to the `Model` folder with the following content:

```
// Model/UserProfileViewModel.cs

namespace Auth0Mvc.ViewModels;

public class UserProfileViewModel
{
  public string EmailAddress { get; set; }

  public string Name { get; set; }

  public string ProfileImage { get; set; }
}
```

This code simply defines a class with the three elements of the user profile: the name, the email address, and the user's picture.

Create the profile view

To define the associated view, create an `Account` folder in the `Views` folder. Then add the `Profile.cshtml` file to the `Views/Account` folder with the following content:

auth0.com | © Auth0 2022

```
@* Views/Account/Profile.cshtml *@

@model Auth0Mvc.ViewModels.UserProfileViewModel
@{
    ViewData[“Title”] = “User Profile”;
}

<div class=”row”>
    <div class=”col-md-12”>
        <div class=”row”>
            <h2>@ViewData[“Title”].</h2>

            <div class=”col-md-2”>
                <img src=”@Model.ProfileImage”
                     alt=”” class=”img-rounded img-responsive” />
            </div>
            <div class=”col-md-4”>
                <h3>@Model.Name</h3>
                <p>
                    <i class=”glyphicon glyphicon-envelope”></i>
                    @Model.EmailAddress
                </p>
            </div>
        </div>
    </div>
</div>
```

This markup shows the three properties of the user model.

Create the profile controller

Let’s complete the profile by creating the profile controller. Add the following method to the `AccountController` class:

auth0.com | © Auth0 2022

```
// Controllers/AccountController.cs

// ...existing code...
using System.Security.Claims;  // 👈 new code
using Auth0Mvc.ViewModels;    // 👈 new code

public class AccountController : Controller
{
  // ...existing code...

  // 👇 new code
  [Authorize]
  public IActionResult Profile()
  {
    return View(new UserProfileViewModel()
    {
      Name = User.Identity.Name,
      EmailAddress = User.FindFirst(c => c.Type == ClaimTypes.Email)?.Value,
      ProfileImage = User.FindFirst(c => c.Type == "picture")?.Value
    });
  }
  // ☝️ new code
}
```

You added the references to the `System.Security.Claims` and `Auth0Mvc.ViewModels` namespaces and defined the `Profile()` method. This method returns a view based on the user profile model, whose property values are extracted by the currently logged `User` object. Notice also that the `Profile()` method is marked with the `Authorized` attribute.

auth0.com | © Auth0 2022

Link the profile page

Finally, let's make the profile page available to the user. Open the `_Layout.cshtml` file under the `Views/Shared` folder and update its content as follows:

```
// Views/Shared/_Layout.cshtml

<!DOCTYPE html>
<html lang="en">

<!-- ...existing code -->

  <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
    @if (User.Identity.IsAuthenticated)
    {
      <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
          <a class="nav-link text-dark"
             asp-area=""
             asp-controller="Home"
             asp-action="Index">Home</a>
        </li>
        <li class="nav-item">
          <a class="nav-link text-dark"
             asp-area=""
             asp-controller="Home"
             asp-action="Privacy">Privacy</a>
        </li>
      </ul>

      <ul class="navbar-nav ms-auto">
        // 👇 new code
        <li class="nav-item">
```

auth0.com | © Auth0 2022

```
            <a class="nav-link text-dark"
               asp-controller="Account"
               asp-action="Profile">Hello @User.Identity.Name!</a>
          </li>
          // 👆 new code
          <li class="nav-item">
            <a class="nav-link text-dark"
               asp-area=""
               asp-controller="Account"
               asp-action="Logout">Logout</a>
          </li>
        </ul>
      } else {
        <ul class="navbar-nav ms-auto">
          <li class="nav-item">
            <a class="nav-link text-dark"
               asp-area=""
               asp-controller="Account"
               asp-action="Login">Login</a>
          </li>
        </ul>
        }
    </div>

  <!-- ...existing code -->//
```

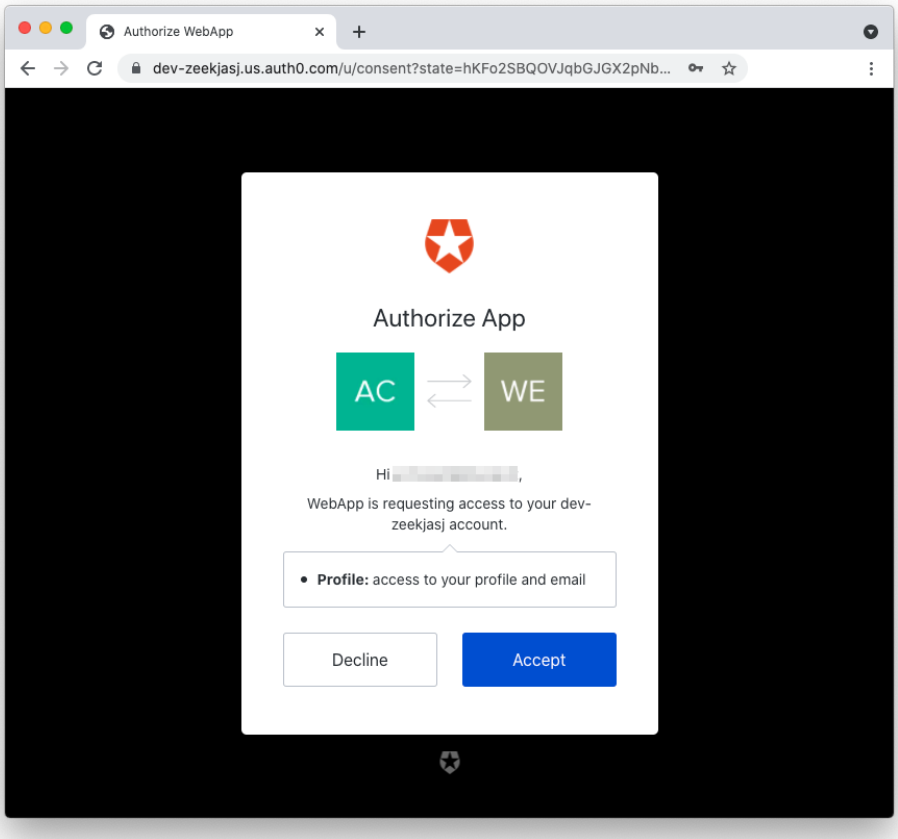
You add the link to the profile view next to the logout link. That link will show a welcome message with the user name. Also, it will be shown only when the user is authenticated.

auth0.com | © Auth0 2022

Test the user profile

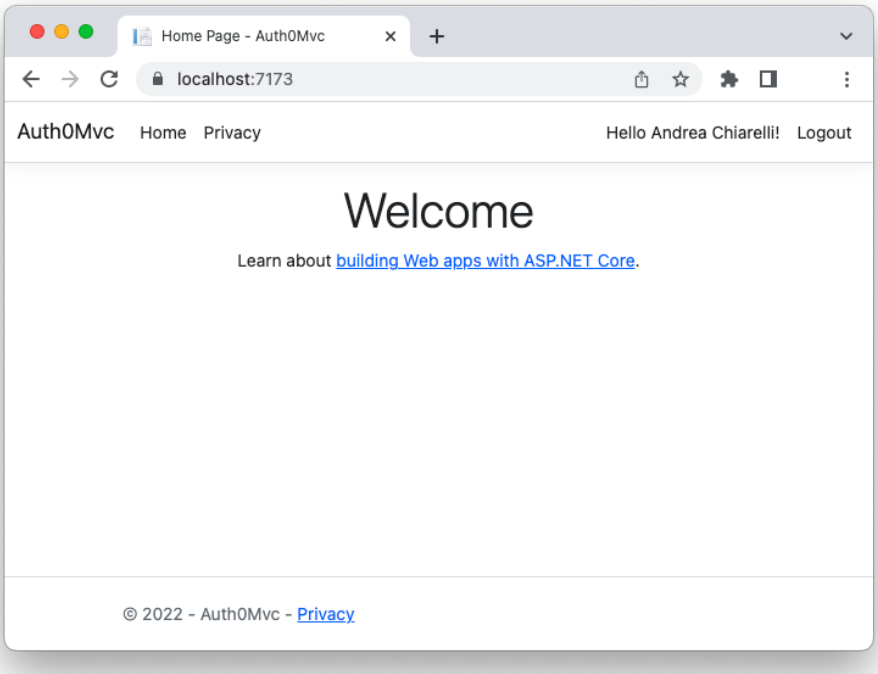
Go back to your browser and authenticate again. As a first difference, you will notice that you will see the consent screen again. Why? Didn't you previously accept this?

Actually, this time your application is requesting a new piece of information about you: the email. If you compare this screen with the previous one, you will notice this subtle difference:

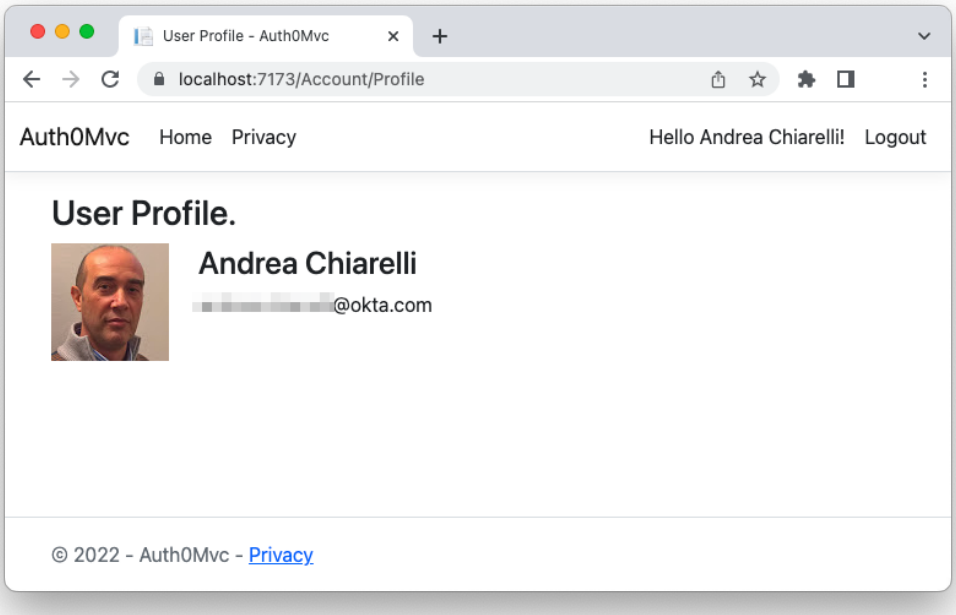


After you accept that screen, your new home page will look as follows:

© Auth0 2022 | auth0.com



You can see the welcome message next to the logout link. If you click the welcome message, the user profile will be shown as in the following picture:



auth0.com | © Auth0 2022

Summary

Throughout this section, you learned how to integrate authentication in an ASP.NET Core MVC application via Auth0. You've seen how the Auth0 ASP.NET Core Authentication SDK handles OpenID Connect for you under the hood, preventing you from dealing with the technical details. You also learned how to protect the private views of your application and how to implement logout. Finally, you were able to get the user's data to create a user profile page.

You can find the complete code of the ASP.NET Core MVC project in the `Auth0Mvc` folder of **this GitHub repository**.

Razor Pages Applications

When it comes to building web applications with ASP.NET, you will find yourself having to **choose between several programming models**. Putting aside Single-Page Applications (SPA) and focusing only on the most recent versions of .NET, you have two programming models for creating traditional web applications: ASP.NET Core MVC and Razor Pages. Which programming model should you choose for your web application?

The ASP.NET Core MVC model is more popular than the Razor Pages model, maybe because ASP.NET Core MVC has a longer history that started in 2009 with ASP.NET MVC. However, this doesn't mean this programming model is better than the Razor Pages model.

Both programming models rely on the same template engine, Razor. However, ASP.NET Core MVC promotes the **Model-View-Controller (MVC) design pattern**, while Razor Pages applications propose a lighter and more page-focused approach. So, when you have to choose which programming model to use for your web application, you should carefully evaluate where your application's behavior fits. As **Microsoft documentation** states,

auth0.com | © Auth0 2022

"if your ASP.NET MVC app makes heavy use of views, you may want to consider migrating from actions and views to Razor Pages".

That said, if you have experience in using both programming models, you may note that the border between the two models is not so neat. Since both models share the same template engine, you may find ASP.NET Core MVC applications that use Razor Pages when a simple page is needed. On the other side, you may find Razor Pages applications that use controllers for functionalities where they make sense.

Mixing the two models lets you use the best of both to build efficient web applications.

To learn more about Razor Pages, check out the **official documentation**. This section will focus on securing a Razor Pages application with Auth0.

The sample application

To learn how to integrate a Razor Pages application with Auth0, you'll start with a simple app built using the standard .NET template. In a terminal window, run the following command:

```
dotnet new razor -o Auth0RazorPages
```

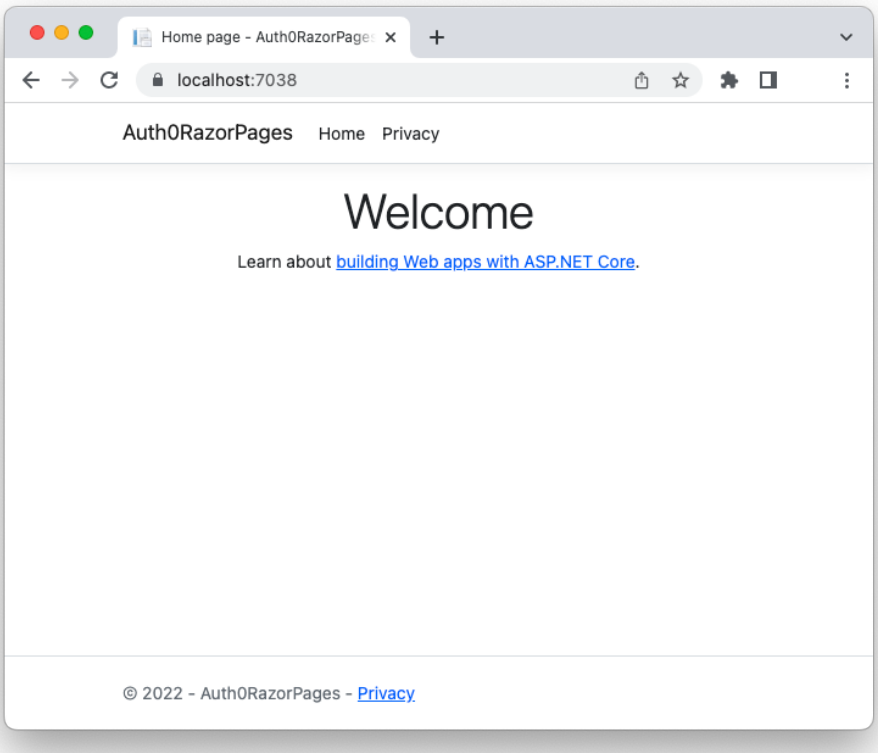
After the command creates the Razor Pages application, go to the `Auth0RazorPages` folder and launch the application with the following command:

```
dotnet run
```

> If you are using a Mac, you may be affected by a known issue when running an ASP.NET Core application through the .NET CLI. Please, refer to **this note** to learn about a workaround.

From your terminal, take note of the address your application is listening to and navigate to it with your browser. In the example shown in this section, the application's address is `https://localhost:7038`. Replace this address with yours wherever it's needed.

If everything works as expected, you should see the following page:



Now you are ready to integrate your application with Auth0.

Register with Auth0

As in the ASP.NET Core MVC case, you start by registering your Razor Pages application with Auth0. The steps are the same, since we are dealing with regular web applications. So, in your Auth0 dashboard, go to the **Applications section** and register the new application. Take note of your Auth0 domain and your client id, and assign the value `https://localhost:<YOUR_PORT_NUMBER>/callback` to the *Allowed Callback*

URLs field and the value `https://localhost:<YOUR_PORT_NUMBER>/` to the *Allowed Logout URLs* field. Don't forget to replace the `<YOUR_PORT_NUMBER>` placeholder with the actual port number assigned to your application.

In the root folder of your Razor Pages application, open the `appsettings.json` configuration file and replace its content with the following:

```
// appsettings.json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Auth0": {
    "Domain": "YOUR_DOMAIN",
    "ClientId": "YOUR_CLIENT_ID"
  }
}
```

Replace the placeholders `YOUR_DOMAIN` and `YOUR_CLIENT_ID` with the respective values taken from the Auth0 dashboard.

Add authentication

To add authentication, you need to apply some changes to your application and use the settings you already stored in the `appsettings.json` file.

auth0.com | © Auth0 2022

Install the Auth0 SDK

Start by installing the **Auth0 ASP.NET Core Authentication SDK**. Run the following command in your terminal window:

```
dotnet add package Auth0.AspNetCore.Authentication
```

As you already know, the Auth0 ASP.NET Core Authentication SDK lets you easily integrate **OpenID Connect-based** authentication in your app without dealing with all its low-level details.

Set up authentication

Now, let's modify the application code to support authentication. Open the `Program.cs` file and change its content as shown below:

```
// Program.cs

using Auth0.AspNetCore.Authentication; // 👈 new code

var builder = WebApplication.CreateBuilder(args);

// 👇 new code
builder.Services
    .AddAuth0WebAppAuthentication(options => {
      options.Domain = builder.Configuration["Auth0:Domain"];
      options.ClientId = builder.Configuration["Auth0:ClientId"];
    });
// 👆 new code


// ...existing code...
```

auth0.com | © Auth0 2022

```
app.UseRouting();

app.UseAuthentication(); // 👈 new code
app.UseAuthorization();

// ...existing code...
```

Following the highlighted code, you added a reference to the `Auth0.AspNetCore.Authentication` namespace at the beginning of the file. Then you invoked the `AddAuth0WebAppAuthentication()` method with the Auth0 domain and client id as arguments. Finally, you called the `UseAuthentication()` method to enable the authentication middleware. Make sure to call `UseAuthentication()` **before** `UseAuthorization()`.

These changes prepare support for authentication via Auth0.

Implement login

To implement login, go to the root folder of the project and add a new Razor page by running the following command in a terminal window:

```
dotnet new page --name Login --namespace Auth0RazorPages.Pages --output Pages/Account
```

This command will create an `Account` folder within the `Pages` folder and add two files there, `Login.cshtml` and `Login.cshtml.cs`. Open the `Login.cshtml.cs` file and replace its content with the following:

```
// Pages/Account/Login.cshtml.cs

using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Authentication;
using Auth0.AspNetCore.Authentication;
```

auth0.com | © Auth0 2022

```
namespace Auth0RazorPages.Pages;

public class LoginModel : PageModel
{
  public async Task OnGet(string returnUrl = "/")
  {
    var authenticationProperties = new LoginAuthenticationPropertiesBuilder()
                .WithRedirectUri(returnUrl)
                .Build();

    await HttpContext.ChallengeAsync(Auth0Constants.AuthenticationScheme,
    authenticationProperties);
  }
}
```

This class defines the page model for the login page. Actually, this page isn't displayed at all. It just creates a set of authentication properties required for the login and triggers the authentication process via Auth0.

To enable this login operation, you need to change the UI. So, open the `_Layout.cshtml` file under the `Pages/Shared` folder and update its content as follows:

```
@* Pages/Shared/_Layout.cshtml *@

<!DOCTYPE html>
<html lang="en">

<!-- ...existing code -->

  <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
    // 👇 new code
    @if (User.Identity.IsAuthenticated)
    {
```

auth0.com | © Auth0 2022

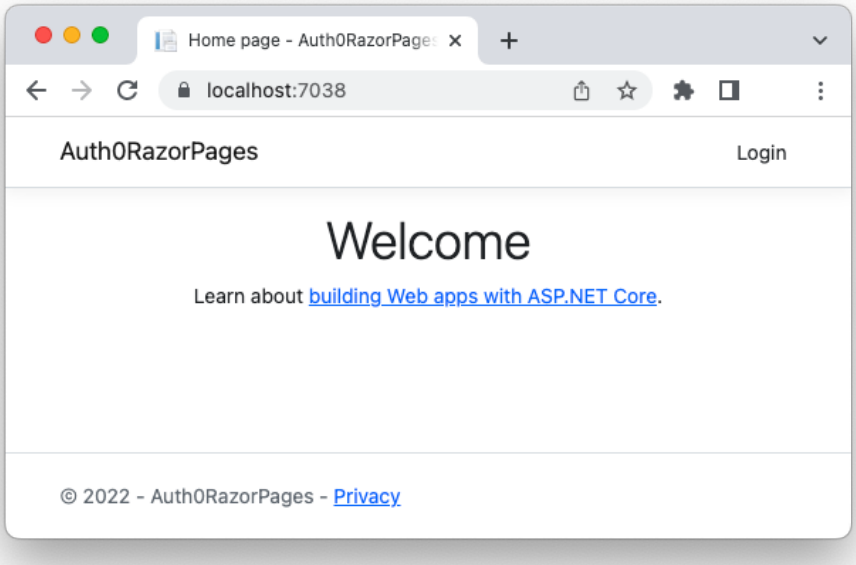
```
      // 👆 new code
        <ul class="navbar-nav flex-grow-1">
          <li class="nav-item">
            <a class="nav-link text-dark"
               asp-area=""
               asp-page="/Index">Home</a>
          </li>
          <li class="nav-item">
            <a class="nav-link text-dark"
               asp-area=""
               asp-page="/Privacy">Privacy</a>
          </li>
        </ul>
      // 👇 new code
      } else {
        <ul class="navbar-nav ms-auto">
          <li class="nav-item">
            <a class="nav-link text-dark" asp-area=""
              asp-page="Account/Login">Login</a>
          </li>
        </ul>
      }
      // 👆 new code
    </div>

<!-- ...existing code -->
```

You modify the page's markup by adding a check on the `User.Identity.IsAuthenticated` property. As you may have figured out, this property lets you know whether the current user is authenticated or not. If the user is not authenticated, a *Login* link will be shown on the right side of the navigation bar.

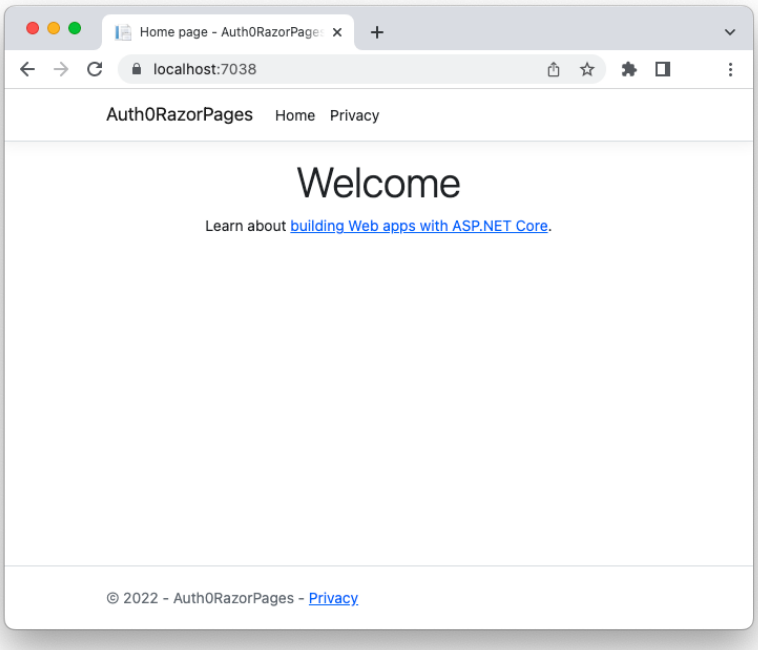
auth0.com | © Auth0 2022

Test login

Everything is ready. Relaunch your app and refresh the page in your browser. You should see the *Login* link as shown in the following picture:



This time, you can't see the *Home* and *Privacy* links on the left side of the home page. You will see them only if you are authenticated. When you click the *Login* link, you will be redirected to the **Auth0 Universal Login page**. After entering the user's credentials and accepting the consent screen, you are redirected back to the home page, which will be shown as before:



auth0.com | © Auth0 2022

Your Razor Pages application now supports authentication!

Protect private pages

Although you have to authenticate to see the *Home* and the *Privacy* links in the navigation bar, the Privacy page itself is not protected. You can verify this by accessing the `https://localhost:7038/Privacy` address directly.

You need to restrict access to this page to authenticated users.

Protect the *Privacy* page

To do this, edit the `Program.cs` file in the root folder of the project as shown below:

```
// Program.cs

using Auth0.AspNetCore.Authentication;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddAuth0WebAppAuthentication(options => {
      options.Domain = builder.Configuration["Auth0:Domain"];
      options.ClientId = builder.Configuration["Auth0:ClientId"];
    });

// Add services to the container.
// 👇 changed code
builder.Services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Privacy");
});
// ☝️ changed code
```

auth0.com | © Auth0 2022

```
var app = builder.Build();

//...existing code...//
```

You used Razor Pages authorization conventions to protect the *Privacy* page from unauthorized access. This approach allows you to centralize access control in your Razor Pages application. Check out the official documentation to learn more about **Razor Pages authorization conventions**.

Now, if you try to access the *Privacy* page before authenticating, you are redirected to the Auth0 Universal Login page.

Implement logout

Another item you need to make your application usable is logout. In fact, currently, when you log in to the application, you stay logged in until your session expires. You may want to allow users to explicitly log out of the application. To do this, add a *Logout* page to the `Account` folder by running the following command:

```
dotnet new page --name Logout --namespace Auth0RazorPages.Pages --output
Pages/Account
```

Then, open the `Logout.cshtml.cs` file and replace its content with the following:

```
// Pages/Account/Logout.cshtml.cs

using Microsoft.AspNetCore.Mvc.RazorPages;
using Auth0.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;
```

auth0.com | © Auth0 2022

```
namespace Auth0RazorPages.Pages
{
    public class LogoutModel : PageModel
    {
        public async Task OnGet()
        {
    var authenticationProperties = new LogoutAuthenticationPropertiesBuilder()
        .WithRedirectUri("/")
        .Build();

    await HttpContext.SignOutAsync(Auth0Constants.AuthenticationScheme,
    authenticationProperties);
    await HttpContext.SignOutAsync(CookieAuthenticationDefaults.
    AuthenticationScheme);
        }
    }
}
```

Like the login page, the logout page doesn't display anything either. It just creates a set of required properties and triggers the logout process to destroy both the Auth0 session and the local session.

The next step is to make the logout link available to the user. So, update the content of the `_Layout.cshtml` file under the `Pages/Shared` folder as follows:

```
@* Pages/Shared/_Layout.cshtml *@

<!DOCTYPE html>
<html lang="en">

<!-- ...existing code -->
```

auth0.com | © Auth0 2022

```
    <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
      @if (User.Identity.IsAuthenticated)
      {
        <ul class="navbar-nav flex-grow-1">
          <li class="nav-item">
            <a class="nav-link text-dark"
                asp-area=""
                asp-page="/Index">Home</a>
          </li>
          <li class="nav-item">
            <a class="nav-link text-dark"
                asp-area=""
                asp-page="/Privacy">Privacy</a>
          </li>
        </ul>
        // 👇 new code
        <ul class="navbar-nav ms-auto">
          <li class="nav-item">
            <a class="nav-link text-dark" asp-area=""
               asp-page="/Account/Logout">Logout</a>
           </li>
         </ul>
        // ☝️ new code
      } else {
        <ul class="navbar-nav ms-auto">
          <li class="nav-item">
            <a class="nav-link text-dark" asp-area=""
                asp-page="Account/Login">Login</a>
          </li>
        </ul>
        }
    </div>

<!-- ...existing code -->
```

auth0.com | © Auth0 2022

You just added the logout link on the right side of the navigation bar. It will be visible when the user is authenticated. That link points to the logout page in the `Account` folder.

As a final step, open the `Program.cs` file and configure the logout page as a protected page, as shown below:

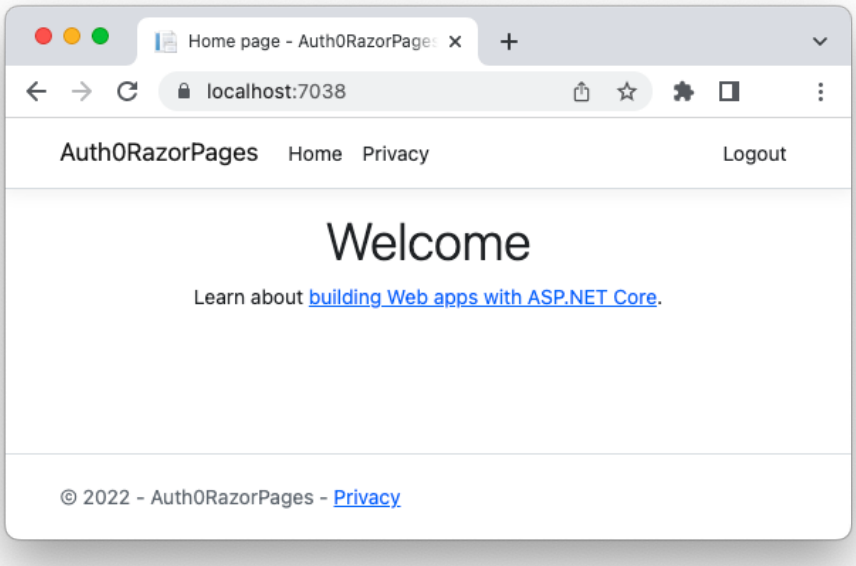
```
// Program.cs

//...existing code...

// Add services to the container.
builder.Services.AddRazorPages(options =>
{
  options.Conventions.AuthorizePage("/Catalog");
  // 👇 new code
  options.Conventions.AuthorizePage("/Account/Logout");
  // 👆 new code
});

//...existing code...
```

Test logout

It's time to test this new version of the application. First of all, **delete all cookies from your browser or use an incognito window**. This is needed because you haven't had a chance to log out so far.

Now, after you log in again, the home page should look as follows:

auth0.com | © Auth0 2022



When you click the *Logout* link, you will be disconnected from Auth0 and see the usual home page with only the *Login* link.

Add a user profile page

Let's try to go one step further and add a page showing some data about the user.

Specify your scopes

Thanks to the Auth0 ASP.NET Core Authentication SDK, your Razor Pages application uses OpenID Connect (OIDC) to authenticate your users. It receives an ID token from Auth0 containing some basic data about the user. From a technical point of view, this user data is available because the SDK requests the `openid` and `profile` scopes by default. You don't see this because the SDK takes care of the whole authentication process and manages the ID token. So, by default, you have the user's name and possibly their picture. If you also want the user's email in their profile, you have to specify the scopes explicitly.

Open the `Program.cs` file and apply the following change:

auth0.com | © Auth0 2022

```
// Program.cs

//...existing code...

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddAuth0WebAppAuthentication(options => {
      options.Domain = builder.Configuration["Auth0:Domain"];
      options.ClientId = builder.Configuration["Auth0:ClientId"];
      options.Scope = "openid profile email"; // 👈 new code
    });

//...existing code...
```

You assigned a string to the `Scope` option containing the default scopes mentioned previously (`openid` and `profile`) and the `email` scope.

This causes the email to be added to the user profile.

Create the profile page

Now, let's create a new page to show the user's profile data. From the root folder of the project, run the following command:

```
dotnet new page --name Profile --namespace Auth0RazorPages.Pages --output
Pages/Account
```

As usual, it creates two new files in the `Pages/Account` folder: `Profile.cshtml` and `Profile.cshtml.cs`.

Open the `Profile.cshtml.cs` file and replace its content with the following:

auth0.com | © Auth0 2022

```
// Pages/Account/Profile.cshtml.cs

using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Security.Claims;

namespace Auth0RazorPages.Pages;

public class ProfileModel : PageModel
{
  public string UserName { get; set; }
  public string UserEmailAddress { get; set; }
  public string UserProfileImage { get; set; }
  public void OnGet()
  {
    UserName = User.Identity.Name;
    UserEmailAddress = User.FindFirst(c => c.Type == ClaimTypes.Email)?.Value;
    UserProfileImage = User.FindFirst(c => c.Type == "picture")?.Value;
  }
}
```

This code simply defines the `ProfileModel` class with the three elements of the user profile: the name, the email address, and the user's picture.

Now, define the associated view by replacing the content of the `Profile.cshtml` file in the same folder with the following content:

```
@* Pages/Account/Profile.cshtml *@

@page
@model Auth0RazorPages.Pages.ProfileModel
@{
    ViewData["Title"] = "User Profile";
}
```

auth0.com | © Auth0 2022

```
<div class="row">
    <div class="col-md-12">
        <div class="row">
            <h2>@ViewData["Title"].</h2>

            <div class="col-md-2">
                <img src="@Model.UserProfileImage"
                     alt="" class="img-rounded img-responsive" />
            </div>
            <div class="col-md-4">
                <h3>@Model.UserName</h3>
                <p>
                    <i class="glyphicon glyphicon-envelope"></i>
                    @Model.UserEmailAddress
                </p>
            </div>
        </div>
    </div>
</div>
```

This markup shows the three properties of the user model.

Protect the profile page

Before moving on to test this new feature, let's make sure that only authorized users can access the user profile page. So, open the `Program.cs` file and configure the profile page as a protected page as shown below:

```
// Program.cs

//...existing code...
```

auth0.com | © Auth0 2022

```
// Add services to the container.
builder.Services.AddRazorPages(options =>
{
  options.Conventions.AuthorizePage("/Catalog");
  options.Conventions.AuthorizePage("/Account/Logout");
  // 👇 new code
  options.Conventions.AuthorizePage("/Account/Profile");
  // ☝️ new code
});

//...existing code...//
```

You just added the profile page to the pages requiring authorization.

Link the profile page

Finally, let's make the profile page available to the user. Open the `_Layout.cshtml` file under the `Pages/Shared` folder and update its content as follows:

```
@* Pages/Shared/_Layout.cshtml *@

<!DOCTYPE html>
<html lang="en">

<!-- ...existing code -->

  <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
    @if (User.Identity.IsAuthenticated)
    {
      <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
          <a class="nav-link text-dark"
```

auth0.com | © Auth0 2022

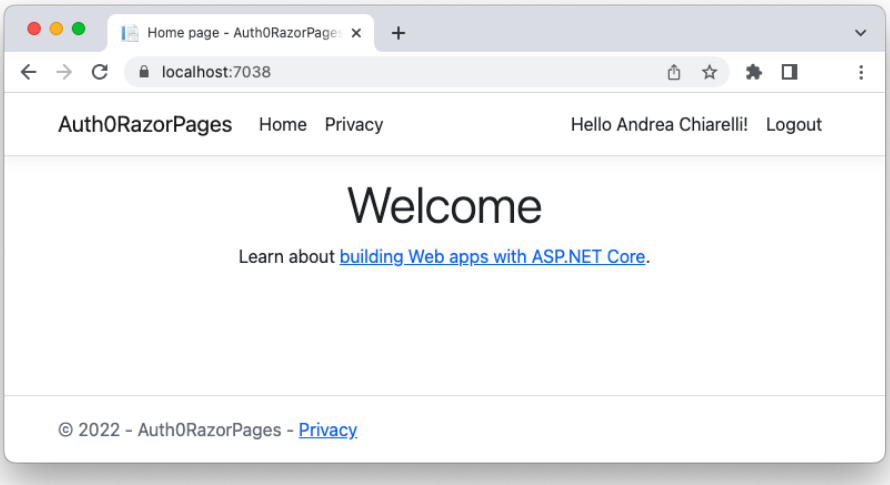
```
                asp-area=""
                asp-page="/Index">Home</a>
          </li>
          <li class="nav-item">
            <a class="nav-link text-dark"
                asp-area=""
                asp-page="/Privacy">Privacy</a>
          </li>
        </ul>
        <ul class="navbar-nav ms-auto">
          // 👇 new code
          <li class="nav-item">
            <a class="nav-link text-dark"
               asp-page="/Account/Profile">Hello @User.Identity.Name!</a>
          </li>
          // 👆 new code
          <li class="nav-item">
            <a class="nav-link text-dark" asp-area=""
               asp-page="/Account/Logout">Logout</a>
           </li>
         </ul>
      } else {
        <ul class="navbar-nav ms-auto">
          <li class="nav-item">
            <a class="nav-link text-dark" asp-area=""
               asp-page="/Account/Login">Login</a>
          </li>
        </ul>
        }
    </div>

<!-- ...existing code -->
```

© Auth0 2022 | auth0.com

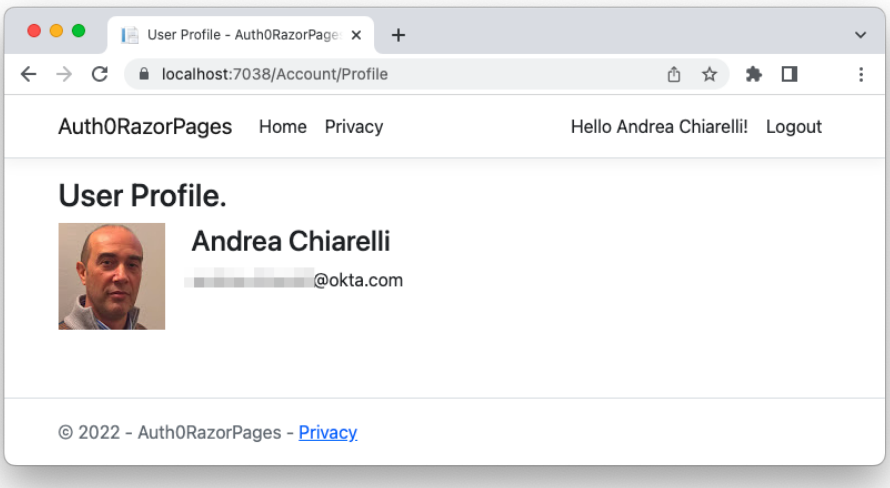
You added the link to the profile view next to the logout link. That link will show a welcome message with the user name. Also, it will be shown only when the user is authenticated.

Test the user profile

Refresh your application and authenticate again. You will notice that you see the consent screen again. Since you added the `email` scope in the OIDC configuration, Auth0 requests the user's authorization to provide this new piece of information about you. After you accept that screen, your new home page will look as follows:



You can see the welcome message next to the logout link. If you click the welcome message, the user profile will be shown as in the following picture:



Summary

Congratulations! You completed your application! Throughout this journey, you learned how to integrate authentication in a Razor Pages application via Auth0. You've seen how the Auth0 ASP.NET Core Authentication SDK handles OpenID Connect for you under the hood, preventing you from dealing with the technical details. You also learned how to protect the private pages of your application and how to implement logout. Finally, you were able to create a user profile page.

You can find the complete code of this Razor Pages project in the `Auth0RazorPages` folder of **this GitHub repository**.

Blazor Server Applications

Blazor has been gaining in popularity, especially after the release of .NET Core 3.0, which enriched it with many interesting features. Subsequent versions of .NET consolidated its foundations, and interest around it is growing so much that Microsoft is betting a lot on its future. But what is Blazor exactly?

Blazor is a programming framework for building web applications with .NET. It allows .NET developers to use their C# and **Razor** knowledge to build interactive UIs running in the browser. Developing web applications with Blazor brings a few benefits to .NET developers:

- They use C# and Razor instead of JavaScript and HTML.
- They can leverage all of .NET's functionalities.
- They can share code across the server and client.
- They can use the .NET development tools they are used to.

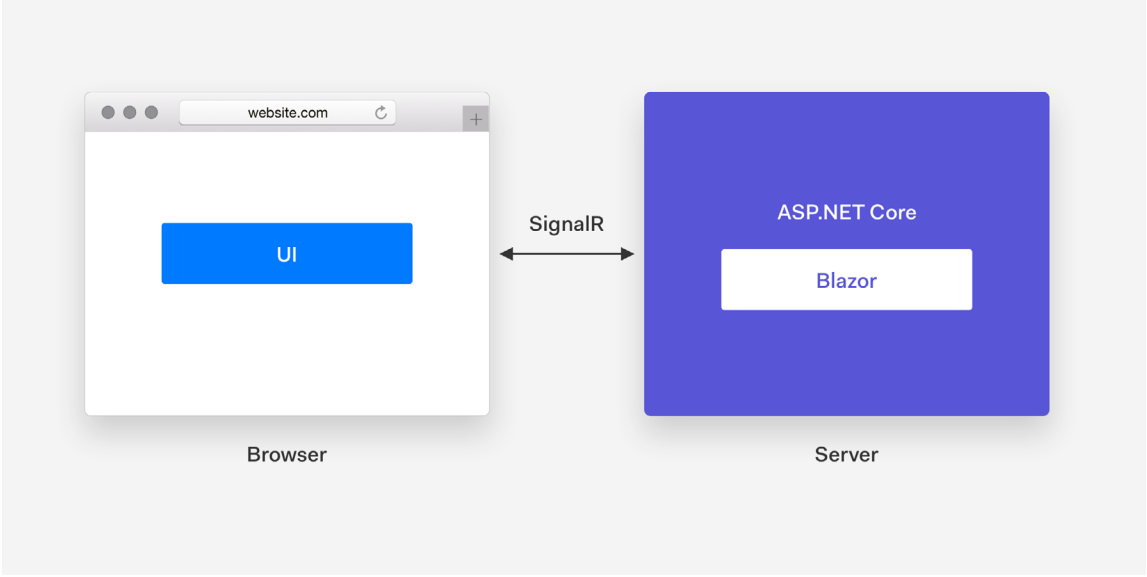
In a nutshell, Blazor promises .NET developers the ability to build web applications with the development platform they are comfortable with.

auth0.com | © Auth0 2022

The server hosting model

Blazor provides two ways to run your web application: *Blazor Server* and *Blazor WebAssembly*. These are called *hosting models*.

The *Blazor Server* hosting model runs your application on the server within an ASP.NET Core application. The UI is sent to the browser, but UI updates and event handling are performed on the server side. This is similar to traditional Web applications, but the communication between the client side and the server side happens over a **SignalR** connection. The following picture gives you an idea of the overall architecture of the Blazor Server hosting model:



The Blazor WebAssembly hosting model, also known as Blazor WASM, lets your application run entirely on the user's browser. You will learn more about Blazor WebAssembly applications and their integration with Auth0 in **Chapter 5**.

The sample application

To get started with the Blazor Server hosting model, you will create a basic application by running the following command in a terminal window:

```
dotnet new blazorserver -o Auth0BlazorServer
```

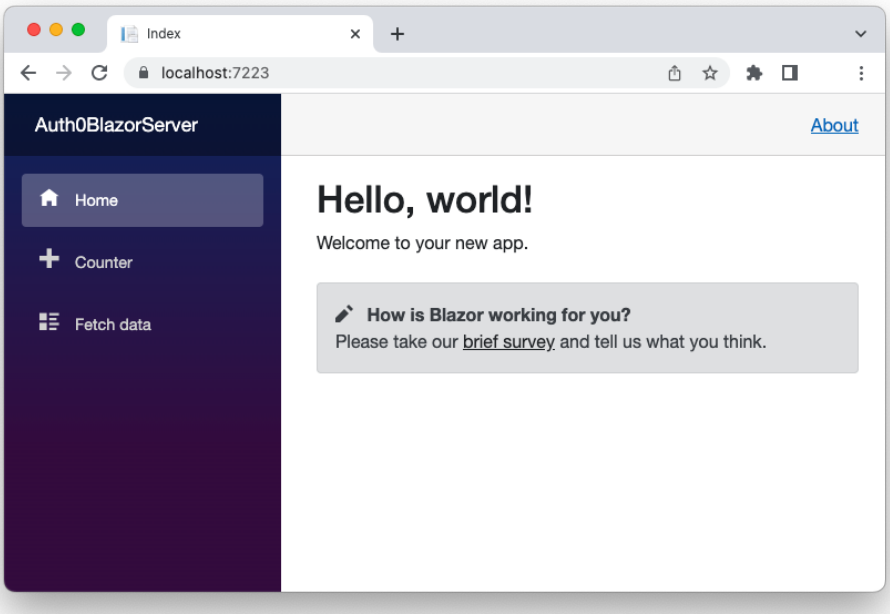
This command uses the `blazorserver` template to generate the project for your application in the `Auth0BlazorServer` folder. To check that everything works as expected, go to that folder and launch the application with the following command:

```
dotnet run
```

> If you are using a Mac, you may be affected by a known issue when running an ASP.NET Core application through the .NET CLI. Please, refer to **this note** to learn about a workaround.

Look at your terminal and take note of the address your application is listening to. Then, navigate to it with your browser. In the example shown in this section, the application's address is `https://localhost:7223`. Replace this address with yours wherever it's needed.

If everything works as expected, you should see the following page:



The application that comes with the Blazor Server project template provides three pages accessible from the menu on the left side:

- The home page that you see in the picture above.
- The *Counter* page, which implements a simple counter incremented by clicking a button. The code of this page demonstrates how to execute code on the client side of the application without involving the server.
- The *Fetch data* page, which shows some fictitious weather forecast data. Its code demonstrates how to implement interaction between the client and server sides of the application.

Let's start integrating this Blazor Server application with Auth0.

Register with Auth0

The first step in securing your Blazor Server application is to access the Auth0 dashboard to register your Auth0 application. Once in the dashboard, move to the **Applications section** and follow the usual steps for a regular web application:

Let's focus on the relevant point of this investigation:

1. Click on *Create Application*.
2. Provide a friendly name for your application (for example, *Blazor Server App*) and select *Regular Web Applications* as the application type.
3. Finally, click the *Create* button.

These steps make Auth0 aware of your Blazor Server application.

After the application has been created, go to the *Settings* tab and take note of your Auth0 domain and client id. Then, in the same form, assign the value `https://localhost:<YOUR_PORT_NUMBER>/callback` to the *Allowed Callback URLs* field and the value `https://localhost:<YOUR_`

PORT_NUMBER>/ to the *Allowed Logout URLs* field. Replace the <YOUR_PORT_NUMBER> placeholder with the actual port number assigned to your application.

Click the *Save Changes* button to your changes.

Now, open the appsettings.json file in the root folder of your Blazor Server project and replace its content with the following:

```
// appsettings.json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Auth0": {
    "Domain": "YOUR_DOMAIN",
    "ClientId": "YOUR_CLIENT_ID"
  }
}
```

Replace the placeholders YOUR_DOMAIN and YOUR_CLIENT_ID with the respective values taken from the Auth0 dashboard.

Add authentication

Let's start the Auth0 integration by following similar steps as for ASP.NET Core MVC and Razor Pages applications.

Install the Auth0 SDK

Install the **Auth0 ASP.NET Core Authentication SDK** by running the following command in your terminal window:

```
dotnet add package Auth0.AspNetCore.Authentication
```

As you already know, the Auth0 ASP.NET Core SDK lets you easily integrate **OpenID Connect**-based authentication in your app without dealing with all its low-level details.

Set up authentication

After the installation is complete, open the `Program.cs` file and change its content as follows:

```
// Program.cs

using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;
using Auth0BlazorServer.Data;
using Auth0.AspNetCore.Authentication; // 👈 new code

var builder = WebApplication.CreateBuilder(args);

// 👇 new code
builder.Services
    .AddAuth0WebAppAuthentication(options => {
      options.Domain = builder.Configuration["Auth0:Domain"];
      options.ClientId = builder.Configuration["Auth0:ClientId"];
    });
// ☝️ new code
```

```
// Add services to the container.
builder.Services.AddRazorPages();

// ...existing code...

app.UseRouting();

app.UseAuthentication(); // 👈 new code
app.UseAuthorization();  // 👈 new code

app.MapBlazorHub();
app.MapFallbackToPage("/_Host");

app.Run();
```

Following the highlighted code, you added a reference to the `Auth0.AspNetCore.Authentication` namespace at the beginning of the file. Then you invoked the `AddAuth0WebAppAuthentication()` method with the Auth0 domain and client id as arguments. These Auth0 configuration parameters are taken from the `appsetting.json` configuration file you prepared earlier. Finally, you called the `UseAuthentication()` and `UseAuthorization()` methods to enable the authentication and authorization middleware.

Your application now has the infrastructure to support authentication via Auth0.

Implement login

To let users authenticate, you need to redirect them to the Auth0 Universal Login page. In the Blazor Server hosting model, the communication between the client and the server does not occur over HTTP, but through SignalR. Since Auth0 uses standard protocols such as **OpenID Connect**

auth0.com | © Auth0 2022

and **OAuth2** that rely on HTTP, you need to provide a way to bring those protocols on Blazor.

To solve this issue, you are going to create an endpoint, `/login`, that redirects authentication requests to Auth0. A standard Razor page responds behind this endpoint. So, add the *Login* razor page to the project by typing the following command in a terminal window:

```
dotnet new page --name Login --namespace Auth0BlazorServer.Pages --output Pages
```

This command creates two files in the `Pages` folder: `Login.cshtml` and `Login.cshtml.cs`.

Open the `Login.cshtml.cs` file in the `Pages` folder and replace its content with the following:

```
// Pages/Login.cshtml.cs

using Microsoft.AspNetCore.Authentication;
using Auth0.AspNetCore.Authentication;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace Auth0BlazorServer.Pages;

public class LoginModel : PageModel
{
  public async Task OnGet(string redirectUri)
  {
    var authenticationProperties = new LoginAuthenticationPropertiesBuilder()
        .WithRedirectUri(redirectUri)
        .Build();

    await HttpContext.ChallengeAsync(Auth0Constants.AuthenticationScheme,
    authenticationProperties);
  }
}
```

auth0.com | © Auth0 2022

This code creates a set of authentication properties required for the login and triggers the authentication process via Auth0.

Now you need to make some changes to the UI to let users authenticate and see different content depending on whether they are logged in or not.

Open the `App.razor` file in the root folder of the project and replace its content with the following markup:

```
@* App.razor *@

<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(App).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@
            typeof(MainLayout)">
                <Authorizing>
                    <p>Determining session state, please wait...</p>
                </Authorizing>
                <NotAuthorized>
                    <h1>Sorry</h1>
                    <p>You're not authorized to reach this page.
                    You need to log in.</p>
                </NotAuthorized>
            </AuthorizeRouteView>
            <FocusOnNavigate RouteData="@routeData" Selector="h1" />
        </Found>
        <NotFound>
            <PageTitle>Not found</PageTitle>
            <LayoutView Layout="@typeof(MainLayout)">
                <p role="alert">Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

auth0.com | © Auth0 2022

Here you are using the `AuthorizeRouteView` component, which displays the associated component only if the user is authorized. In practice, the content of the `MainLayout` component will be shown only to authorized users. If the user is not authorized, they will see the content wrapped by the `NotAuthorized` component. If the authorization is in progress, the user will see the content inside the `Authorizing` component. The `CascadingAuthenticationState` component will propagate the current authentication state to the inner components so that they can work on it consistently.

To allow users to authenticate, create a Razor component by adding an `AccessControl.razor` file to the `Shared` folder with the following content:

```
@* Shared/AccessControl.razor *@


<AuthorizeView>
    <NotAuthorized>
        <a href="login?redirectUri=/">Log in</a>
    </NotAuthorized>
</AuthorizeView>
```

This component uses the `NotAuthorized` component to let unauthorized users access the *Log in* link. This link points to the `login` endpoint you created before. In particular, it specifies the home page as the URI to redirect users to after authentication.

The final step is to put this component in the top bar of your Blazor application. So, replace the content of the `MainLayout.razor` file with the following content:

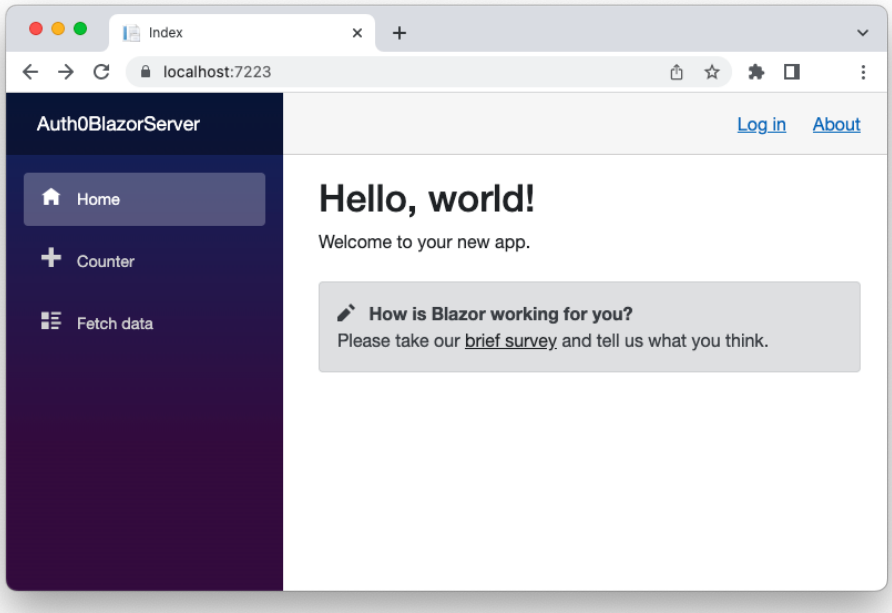
auth0.com | © Auth0 2022

```
@* Shared/MainLayout.razor *@

@inherits LayoutComponentBase

<PageTitle>Auth0BlazorServer</PageTitle>
<div class="page">
    <div class="sidebar">
        <NavMenu />
    </div>

    <main>
        <div class="top-row px-4">
            <AccessControl />    // 👈 new markup
            <a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>
        </div>

        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

As you can see, the only difference is the addition of the `AccessControl` component just before the *About* link.

Test login

Make sure you're in the root folder of the project and run `dotnet run` in your terminal. When a user tries to access your application, they will see the *Log in* link as shown in the following picture:



When they click the *Log in* link, they will be redirected to the **Auth0 Universal Login page**. After entering the user's credentials and accepting the consent screen, they are redirected back to the home page, which will appear as before.

Protect private pages

You added authentication to your Blazor Server application. However, unauthenticated users can still access the *Counter* and *Fetch* data pages. You need to restrict access to these pages only to authenticated users.

Open the `Index.razor` file in the `Pages` folder and add the `Authorize` attribute as shown in the following code snippet:

```
@* Pages/Index.razor *@

@page "/"
@attribute [Authorize]  // 👈 new code

<PageTitle>Index</PageTitle>
```

auth0.com | © Auth0 2022

```
<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />
```

Add the same attribute to the `Counter.razor` component as well:

```
@* Pages/Counter.razor *@

@page "/counter"
@attribute [Authorize]  // 👈 new code

<PageTitle>Counter</PageTitle>

@* ...existing code... *@
```

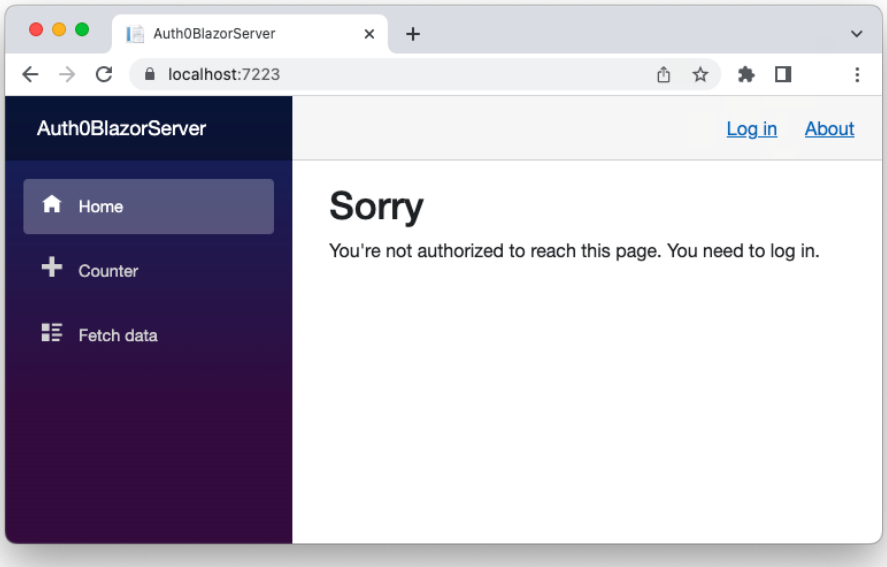
And finally, add that attribute to the `FetchData.razor` component:

```
@* Pages/FetchData.razor *@

@page "/fetchdata"
@attribute [Authorize]  // 👈 new code

<PageTitle>Weather forecast</PageTitle>

@* ...existing code... *@
```

These changes ensure that the server-side rendering of your pages is triggered only by authorized users.

Rerun your application. This time its home page will appear as shown in the following picture:

auth0.com | © Auth0 2022



This message comes from the `<NotAuthorized>` component you configured in the `App.razor` file earlier. The `<AuthorizeRouteView>` component detects that the requested page requires the user to be authorized, but the user is not. You get the same message when you navigate to the other pages. After authentication, your will be able to access the pages as before.

Implement logout

To allow the user to log out of the application, let's create a *Logout* page by typing the following command in the root folder of your Blazor Server project:

```
dotnet new page --name Logout --namespace Auth0BlazorServer.Pages --output Pages
```

You will get two new files in the `Pages` folder: `Logout.cshtml` and `Logout.cshtml.cs`. Replace the content of the `Logout.cshtml.cs` file with the following code:

auth0.com | © Auth0 2022

```
// Pages/Logout.cshtml.cs

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Authentication;
using Auth0.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;

namespace Auth0BlazorServer.Pages;
public class LogoutModel : PageModel
{
  [Authorize]
  public async Task OnGet()
  {
    var authenticationProperties = new LogoutAuthenticationPropertiesBuilder()
         .WithRedirectUri("/")
         .Build();

    await HttpContext.SignOutAsync(Auth0Constants.AuthenticationScheme,
    authenticationProperties);
    await HttpContext.SignOutAsync(CookieAuthenticationDefaults.
    AuthenticationScheme);
  }
}
```

This code closes the user's session on your Blazor application and on Auth0. Notice the `Authorize` attribute attached to the `OnGet()` method. This makes this method available only to authenticated users.

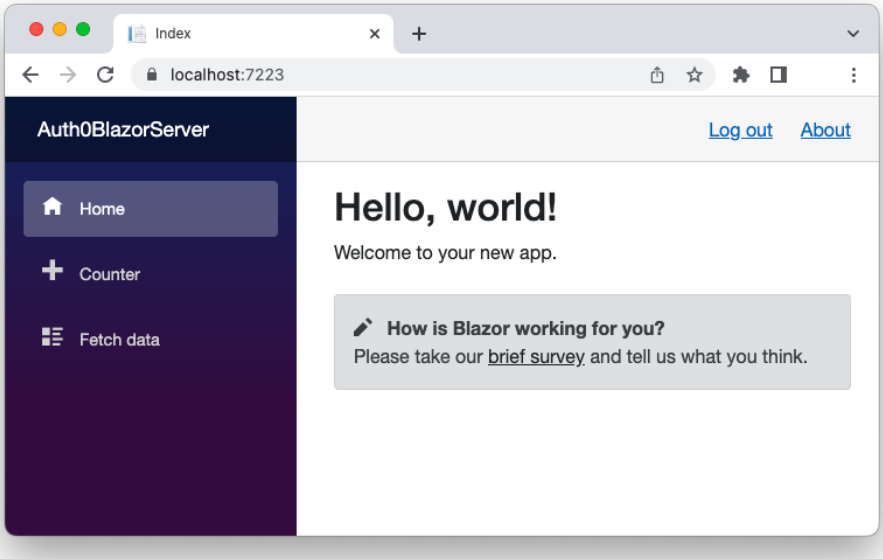
Now, open the `AccessControl.razor` file in the `Shared` folder and add the markup highlighted below:

© Auth0 2022 | auth0.com

```
@* Shared/AccessControl.razor *@

<AuthorizeView>
    // 👇 new markup
    <Authorized>
        <a href="logout">Log out</a>
    </Authorized>
    // 👆 new markup
    <NotAuthorized>
        <a href="login?redirectUri=/">Log in</a>
    </NotAuthorized>
</AuthorizeView>
```

You added the *Log out* link pointing to the `logout` page. That link is wrapped by the `<Authorized>` component, which will make it visible only to authenticated users.

Run your application again and log in. After authentication, you will see the *Log out* link in place of the *Log in* link, as shown below:



© Auth0 2022 | auth0.com

Add a user profile page

Once you add authentication to your Blazor Server application, you may need to access some information about the authenticated user, such as their name and picture. By default, the Auth0 ASP.NET Core Authentication SDK takes care of getting this information for you during the authentication process.

To show the user's name and picture on a page, replace the content of the `Index.razor` component in the `Pages` folder with the following code

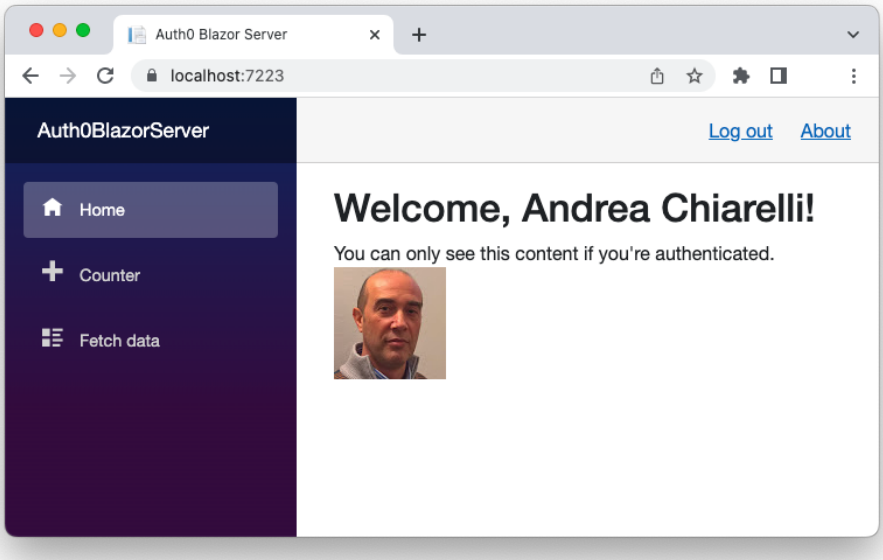
```
@* Pages/Index.razor *@

@page "/"
@inject AuthenticationStateProvider AuthState
@attribute [Authorize]

<PageTitle>Auth0 Blazor Server</PageTitle>

<h1>Welcome, @Username!</h1>
You can only see this content if you're authenticated.
<br />
<img src="@Picture">

@code {
  private string Username = "Anonymous User";
  private string Picture = "";

  protected override async Task OnInitializedAsync()
  {
    var state = await AuthState.GetAuthenticationStateAsync();

    Username = state.User.Identity.Name?? string.Empty;
```

© Auth0 2022 | auth0.com

```
        Picture = state.User.Claims
                    .Where(c => c.Type.Equals("picture"))
                    .Select(c => c.Value)
                    .FirstOrDefault() ?? string.Empty;

        await base.OnInitializedAsync();
    }
}
```

In this new version of the component, you injected the `AuthenticationStateProvider`, which provides you with information about the current authentication state. You get the actual authentication state in the code block by using its `GetAuthenticationStateAsync()` method. Thanks to the authentication state, you can extract the name and the picture of the current user and assign them to the `Username` and `Picture` variables. These are the variables you use in the component's markup to customize this view.

You can now run your application once again, log in, and get a home page similar to the following:



auth0.com | © Auth0 2022

Summary

You started this journey by taking a high-level look at the Blazor framework. Then you created a basic Blazor Server application, registered it with Auth0, and added authentication support using the Auth0 ASP.NET Core Authentication SDK. You learned how to add login and logout functionality to your application and how to protect its pages from unauthorized users. Finally, you added a user profile page showing the name and the picture of the currently logged-in user.

The full source code of the application built throughout this journey can be downloaded from the `Auth0BlazorServer` folder of **this GitHub repository**.

auth0.com | © Auth0 2022

Chapter 4 - APIs

auth0.com | © Auth0 2022

From the OAuth2 point of view, an API is an application that acts as a resource server. This means that the API application is responsible for protecting a user's resource from unauthorized access. If you recall the OAuth flows described in **Chapter 1**, the resource server is involved in checking authorization to access the user's resource. It is not directly involved in user authentication. Basically, it receives an access token from the client and verifies if the client is authorized to access the user's resource. This is the general behavior you will see in the applications described in this chapter.

.NET offers a few ways to create APIs, most of them based on ASP.NET Core. Generally, an API based on the ASP.NET Core framework is called Web API, since it uses the Web technologies stack. .NET provides two approaches to create Web APIs: **ASP.NET Core minimal Web API** and **ASP.NET Core API.**

Let's see how you can integrate these APIs with Auth0.

auth0.com | © Auth0 2022

ASP.NET Core Minimal Web API

Starting from .NET 6.0, you can leverage a simplified syntax to create Web APIs with the ASP.NET Core framework. This simplified approach is known as ASP.NET Core minimal Web API. In this section, you will see how you can integrate this type of API with Auth0 to protect it from unauthorized access.

The sample application

We will keep the application as simple as possible so that we can focus more on the Auth0 integration than the API development details. So, let's start creating a basic ASP.NET Core minimal Web API application by launching the following command in a terminal window:

```
dotnet new webapi -minimal -o Auth0MinimalWebAPI
```

After a few seconds, you will get your project in the `Auth0MinimalWebAPI` folder. Run the project to make sure that everything works as expected by using the following command:

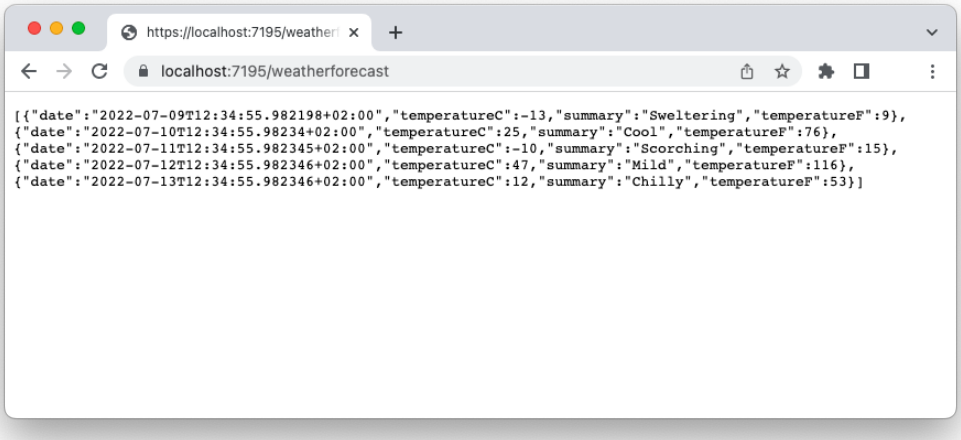
```
dotnet run
```

> If you are using a Mac, you may be affected by a known issue when running an ASP.NET Core application through the .NET CLI. Please, refer to **this note** to learn about a workaround.

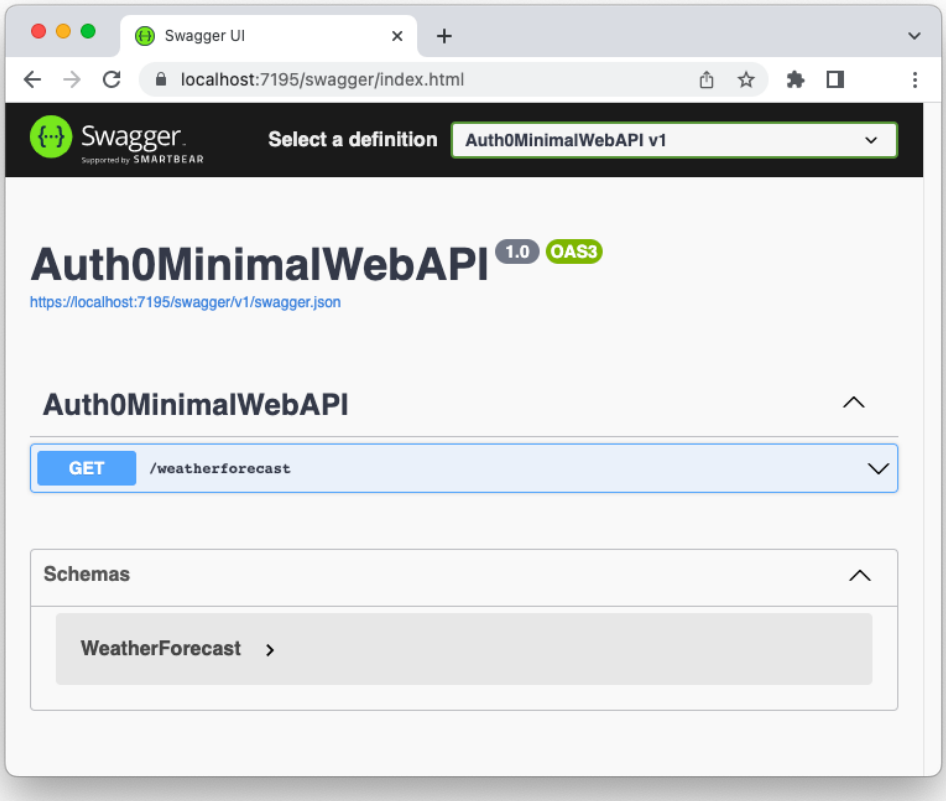
Take note of the address shown in your terminal. This is the base address which your application is listening to. In the examples shown in this section, the application's base address is `https://localhost:7195`. Replace this address with yours wherever it's needed. Navigate to the `weatherforecast` endpoint of your Web API with your browser. In the example shown in this section, the address of this endpoint will be `https://localhost:7195/`

auth0.com | © Auth0 2022

`weatherforecast`. You will see a page similar to the following on your browser:



By default, your API has **Swagger** support already integrated. This allows you to access the API through interactive documentation. You can access this page through the `/swagger` endpoint. You will see the following page:



This page is automatically generated when the application runs in the development environment. It allows you to get information about the endpoints exposed by the Web API. You can also test each endpoint directly from that page. However, to keep things simple when you add authorization support, you will use **curl** to make your requests to the API.

For example, your request to the API will look like the following:

```
curl https://localhost:7195/weatherforecast
```

And you will get an output similar to the one you received in the browser.

Register with Auth0

To start securing your API with Auth0, you need to register it through the Auth0 dashboard. Go to the **API section of the dashboard** and follow these steps:

1. Click on *Create API*.
2. Provide a friendly name for your API (for example, *Minimal Web API*) and a unique identifier (*audience*) in the URL format (for example, `https://minimal-web-api.com`).
3. Leave the signing algorithm as RS256 and click the *Create* button

While you are in the Auth0 dashboard, take note of your Auth0 domain. See **Chapter 2** for more information.

Back in your Web API application, open the `appsettings.json` configuration file and replace its content with the following:

auth0.com | © Auth0 2022

```
// appsettings.json

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Auth0": {
    "Domain": "YOUR_DOMAIN",
    "Audience": "YOUR_UNIQUE_IDENTIFIER"
  }
}
```

Replace the `YOUR_DOMAIN` placeholder with your Auth0 domain and the `YOUR_UNIQUE_IDENTIFIER` placeholder with the value you provided as a unique identifier of your API (`https://minimal-web-api.com`, if you kept the value suggested above).

Add authorization support

To restrict access to the API to authorized clients, your application needs to be able to handle tokens in the **JWT** (*JSON Web Token*) format. You can accomplish this by installing the `Microsoft.AspNetCore.Authentication.JwtBearer` library. Go to your application folder and type the following command in a terminal window:

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

When the installation is complete, open the `Program.cs` file and apply the changes shown below:

auth0.com | © Auth0 2022

```
// Program.cs

using Microsoft.AspNetCore.Authentication.JwtBearer; // 👈 new code

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

// 👇 new code
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
     .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options =>
     {
         options.Authority = $"https://{builder.Configuration["Auth0:Domain"]}";
         options.TokenValidationParameters =
           new Microsoft.IdentityModel.Tokens.TokenValidationParameters
         {
             ValidAudience = builder.Configuration["Auth0:Audience"],
             ValidIssuer = $"{builder.Configuration["Auth0:Domain"]}"
         };
     });

builder.Services.AddAuthorization();
// ☝ new code

var app = builder.Build();

// ...existing code...

app.UseHttpsRedirection();

app.UseAuthentication(); // 👈 new code
app.UseAuthorization();   // 👈 new code

// ...existing code...
```

auth0.com | © Auth0 2022

You add the `Microsoft.AspNetCore.Authentication.JwtBearer` namespace. Also, you add the authentication service by specifying the JWT bearer scheme and providing the authority and audience values from the `appsettings.json` configuration file. Then you add the authorization service and, finally, you add the middleware for authentication and authorization through the `UseAuthentication()` and `UseAuthorization()` methods.

> **Why do you need authentication?**
>
> You learned that APIs are involved only with authorization. They are not directly involved in the authentication process. So, you may wonder why you need to register the authentication middleware in your ASP.NET Web API.
>
> The authentication middleware is needed because it is responsible for creating the security context under which the code is running.
>
> Also, the registration order is important: you need to register the authentication middleware before the authorization middleware.
>
> For more details, check out the **ASP.NET Web API authentication and authorization documentation**.

Secure the API endpoint

So far you have added authorization support to your Web API, but you are not using it to protect your endpoint. You can apply access control to your API endpoint simply by calling the `RequireAuthorization()` method as shown below:

```
// Program.cs

// ...existing code...

app.MapGet("/weatherforecast", () =>
{
```

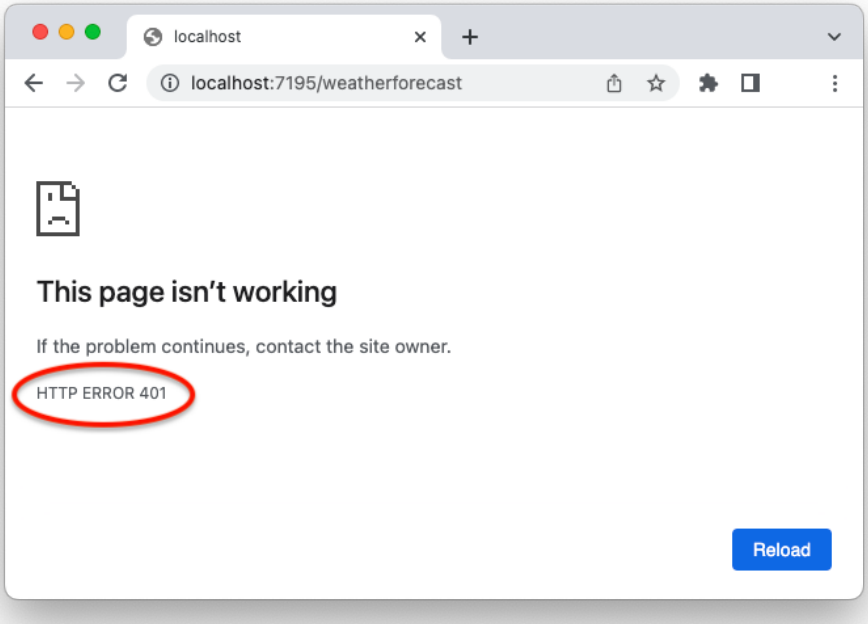
```
    var forecast =  Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateTime.Now.AddDays(index),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)]
        ))
        .ToArray();
    return forecast;
})
.WithName("GetWeatherForecast")
.RequireAuthorization();         // 👈 new code

// ...existing code...
```

This simple change triggers access control when a client makes a request to the `/weatherforecast` endpoint. In practice, your API will check if a valid access token has been provided by the client along with its request.

Test unauthorized access

To make sure that all works as expected, run the application and point your browser to the `/weatherforecast` endpoint. You should get a page like the following:

auth0.com | © Auth0 2022



This time you don't receive the weather forecast data. You receive an error message instead. Specifically, it's a message with the HTTP 401status code, which tells you that you are not authorized.

You get the same response by running the `curl` command as in the following example:

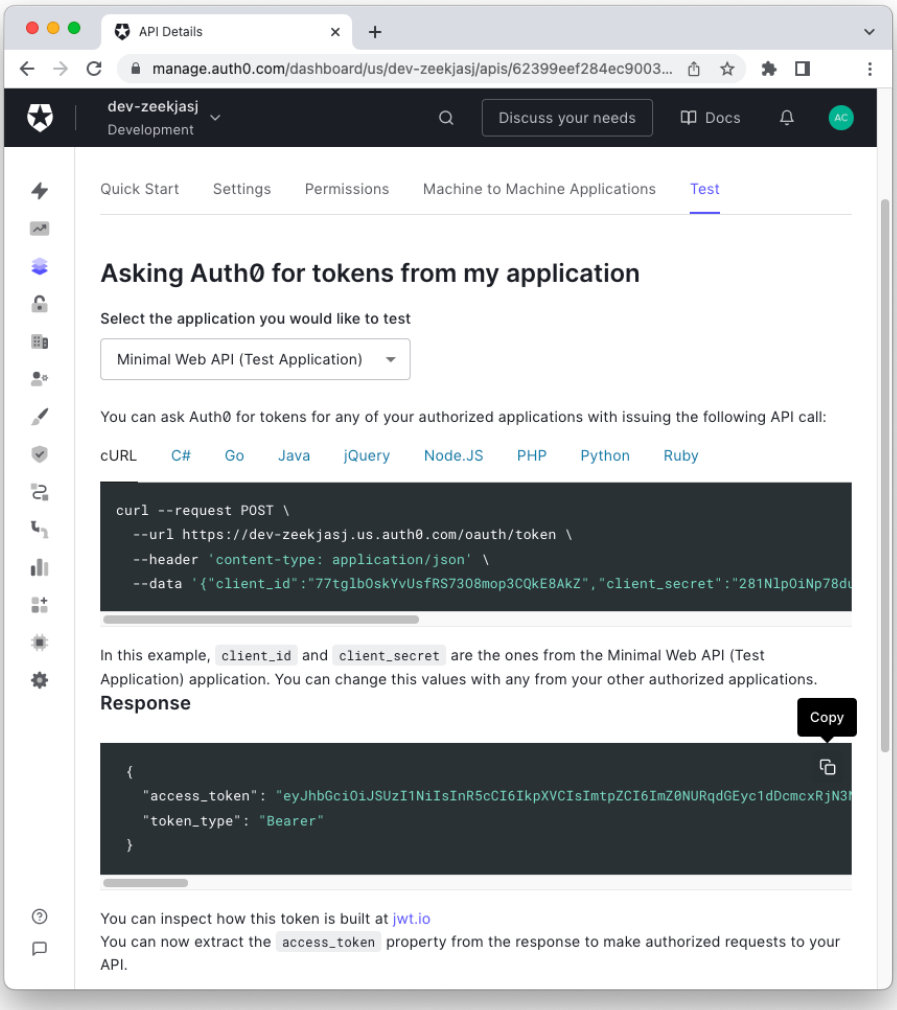
```
curl https://localhost:7195/weatherforecast -v
```

Test authorized access

Once you make sure that unauthorized clients can't access your protected API, let's verify that authorized clients can. To be authorized, your HTTP request must include an access token.

Auth0 allows you to get an access token for testing purposes. Access **the API section of your Auth0 Dashboard** again, select the API that you created before, then select the *Test* tab. In this section, you can get a temporary token to test your Web API by clicking the *Copy Token* icon as shown in the following picture:

auth0.com | © Auth0 2022



Now, head back to your terminal window and run the following command:

```
curl https://localhost:7195/weatherforecast -H 'authorization: bearer
YOUR_ACCESS_TOKEN'
```

Replace the YOUR_ACCESS_TOKEN placeholder with the access token you copied from the Auth0 dashboard. Now you get the weather forecast data again.

Summary

Congratulations! You have protected your ASP.NET Core minimal Web API from unauthorized access.

auth0.com | © Auth0 2022

Starting from a simple working API application, you registered it with Auth0, and built the authorization check support. Then you verified very easily that the API refuses to provide the protected data to unauthorized requests. However, it responds with the proper data to requests that include an access token issued by Auth0.

The full code of the ASP.NET Core minimal Web API project is available in the `Auth0MinimalWebAPI` folder of **this GitHub repository**.

ASP.NET Core Web API

While ASP.NET Core minimal Web APIs offer a streamlined approach to creating Web API in the .NET ecosystem, there are cases in which you need to structure your API project following the classical approach. For example, you may feel this need when the number of endpoints grows, or you come across an existing Web API project. Let's take a look at how you can integrate Auth0 authorization services in your ASP.NET Core Web API application.

The sample application

As usual, the sample application you will use throughout this section is built starting from the standard .NET template. To create it, run the following command in a terminal window:

```
dotnet new webapi -o Auth0WebAPI
```

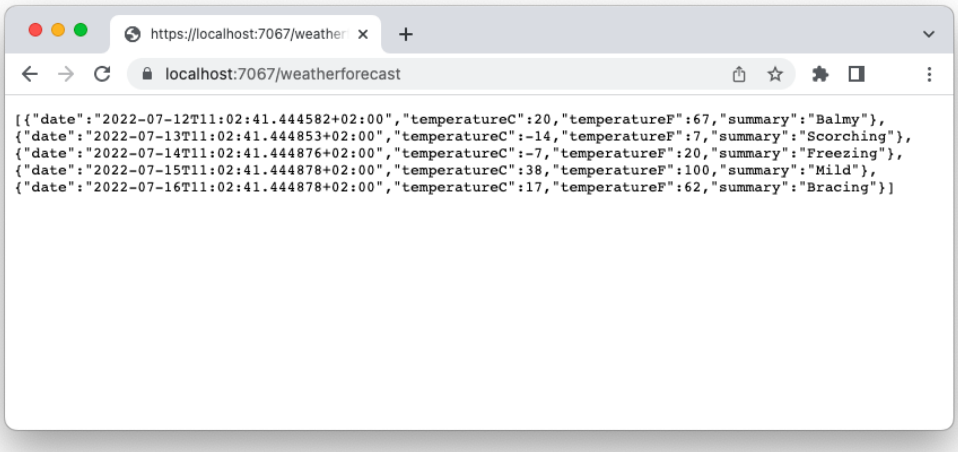
You will get the new project in the `Auth0WebAPI` folder. This project builds the same application you get in the minimal Web API case, but with a different code structure based on controllers. Run the project to make sure that everything works as expected by using the following command:

auth0.com | © Auth0 2022

```
dotnet run
```

> If you are using a Mac, you may be affected by a known issue when running an ASP.NET Core application through the .NET CLI. Please, refer to **this note** to learn about a workaround.

Take note of the base address which your Web API application is listening to. In the example shown in this section, the application's base address is `https://localhost:7067`. Replace this address with yours whenever it's needed. Point your browser to the `/weatherforecast` endpoint of your Web API. In the example shown in this section, the address of this endpoint will be `https://localhost:7067/weatherforecast`. You will see a page similar to the following on your browser:



You also have integrated **Swagger** support in your API. However, to keep things simple with the authorization support you are going to implement, you will use **curl** to make your requests to the API.

For example, your request to the API will look like the following:

```
curl https://localhost:7067/weatherforecast
```

This command will return an output similar to the one you received in your browser.

auth0.com | © Auth0 2022

Register with Auth0

The first step to securing your API with Auth0 is to register it through the Auth0 dashboard. Go to the **API section of the dashboard** and follow these steps:

1. Click on *Create API*.
2. Provide a friendly name for your API (for example, *Web API*) and a unique identifier (*audience*) in the URL format (for example, `https://web-api.com`).
3. Leave the signing algorithm as RS256 and click the *Create* button

Take note of your Auth0 domain — you will need it soon. See **Chapter 2** for more information.

Back in your Web API application, open the `appsettings.json` configuration file and replace its content with the following:

```
// appsettings.json

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Auth0": {
    "Domain": "YOUR_DOMAIN",
    "Audience": "YOUR_UNIQUE_IDENTIFIER"
  }
}
```

auth0.com | © Auth0 2022

Replace the `YOUR_DOMAIN` placeholder with your actual Auth0 domain and the `YOUR_UNIQUE_IDENTIFIER` placeholder with the value you provided as a unique identifier of your API (`https://web-api.com`, if you kept the value suggested above).

Add authorization support

Let's start integrating authorization into your Web API. As you will see, the steps to integrate Auth0 are very similar to the ones you applied to the minimal Web API.

First, install the `Microsoft.AspNetCore.Authentication.JwtBearer` library to handle access tokens in the JWT format. The following command fulfills this task:

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Once the library is installed, open the `Program.cs` file in the root folder of your project and apply the following changes:

```
// Program.cs

using Microsoft.AspNetCore.Authentication.JwtBearer; // 👈 new code

var builder = WebApplication.CreateBuilder(args);

// 👇 new code
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
     .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options =>
     {
         options.Authority = $"https://{builder.Configuration["Auth0:Domain"]}";
         options.TokenValidationParameters =
           new Microsoft.IdentityModel.Tokens.TokenValidationParameters
         {
```

auth0.com | © Auth0 2022

```
            ValidAudience = builder.Configuration["Auth0:Audience"],
            ValidIssuer = $"{builder.Configuration["Auth0:Domain"]}"
        };
    });
// 👆 new code

builder.Services.AddAuthorization();

// ...existing code...

app.UseHttpsRedirection();

app.UseAuthentication(); // 👈 new code
app.UseAuthorization();

app.MapControllers();

app.Run();
```

The first line adds the `Microsoft.AspNetCore.Authentication.JwtBearer` namespace to the current context. Also, you add the authentication service by specifying the JWT bearer scheme and providing the authority and audience values from the `appsettings.json` configuration file. Then you add the middleware for authentication through the `UseAuthentication()` right before the existing `UseAuthorization()` method.

Secure the API endpoint

Now you are ready to secure your Web API endpoint. Open the `WeatherForecastController.cs` file in the `Controllers` folder and change its code as shown below:

auth0.com | © Auth0 2022

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;  // 👈 new code

namespace Auth0WebAPI.Controllers;

[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{

  // ...existing code...

  [HttpGet(Name = "GetWeatherForecast")]
  [Authorize]  // 👈 new code
  public IEnumerable<WeatherForecast> Get()
  {
    return Enumerable.Range(1, 5).Select(index => new WeatherForecast
    {
      Date = DateTime.Now.AddDays(index),
      TemperatureC = Random.Shared.Next(-20, 55),
      Summary = Summaries[Random.Shared.Next(Summaries.Length)]
    })
    .ToArray();
  }
}
```

You added a reference to the `Microsoft.AspNetCore.Authorization` namespace and attached the `Authorize` attribute to the `Get()` method. These changes trigger access control when a client makes a request to the `/weatherforecast` endpoint. In practice, your API will check if a valid access token has been provided by the client along with its request.

auth0.com | © Auth0 2022

Test unauthorized access

Let's verify that these changes work as expected. Run your Web API application and type the following command in a terminal window:

```
curl https://localhost:7067/weatherforecast -v
```

You should get a verbose output with the following relevant section:

```
*
* ...other messages...
*
> GET /weatherforecast HTTP/1.1
> Host: localhost:7067
> User-Agent: curl/7.79.1
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 401 Unauthorized
< Content-Length: 0
< Date: Mon, 11 Jul 2022 10:13:50 GMT
< Server: Kestrel
< WWW-Authenticate: Bearer
<
* Connection #0 to host localhost left intact
```

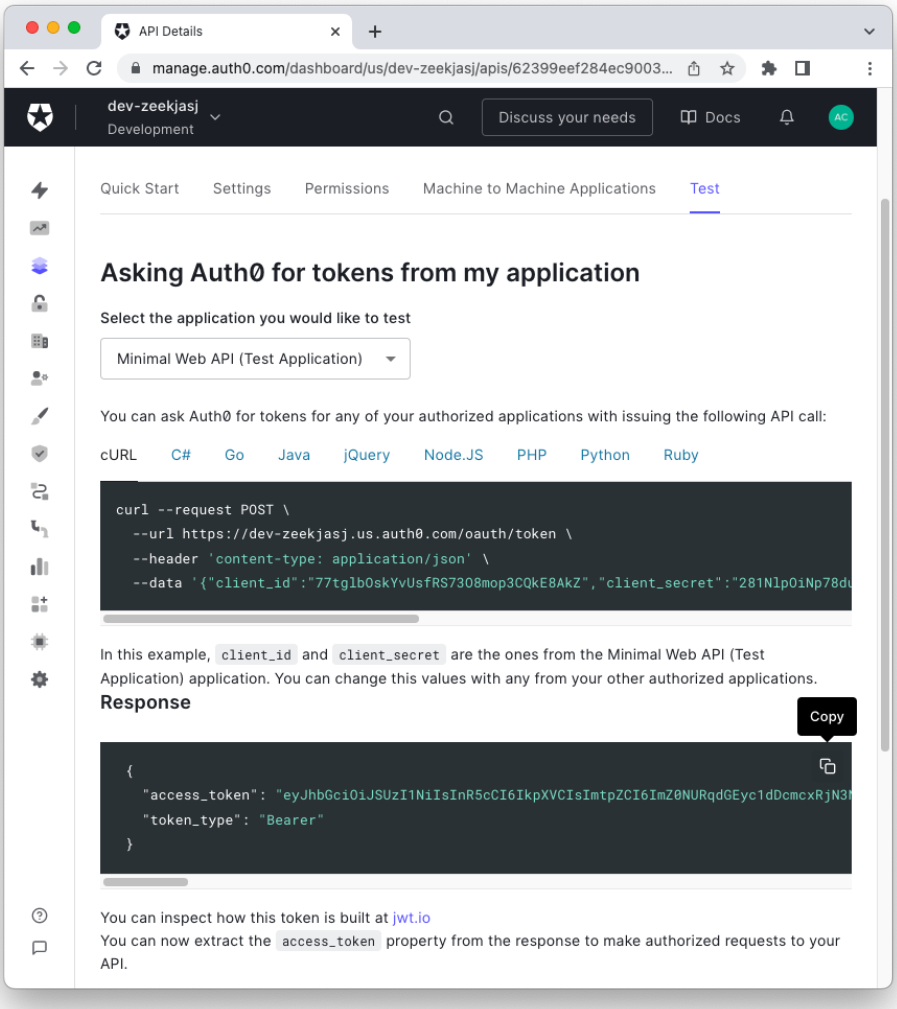
That `401 Unauthorized` status code tells you that your Web API is now protected from unauthorized requests.

Test authorized access

Now let's make sure that authorized clients are allowed to call your API. To be authorized, your HTTP request must include a valid access token. In the

auth0.com | © Auth0 2022

same way as with minimal Web API, you need a test access token. Go to the **API section of your Auth0 Dashboard**, select the API that you created before, then select the *Test* tab. Click the *Copy Token* icon as shown in the following picture to get the access token in your clipboard:



Back in your terminal window, run the following command:

```
curl https://localhost:7067/weatherforecast -H 'authorization: bearer
YOUR_ACCESS_TOKEN'
```

Replace the YOUR_ACCESS_TOKEN placeholder with the access token you copied from the Auth0 dashboard. Now you get the weather forecast data again.

auth0.com | © Auth0 2022

Summary

This section guided you through the integration of an ASP.NET Core Web API with Auth0. As you learned, the process is not so different from the integration of an ASP.NET Core minimal Web API.

You started by registering the application with Auth0 and installed the `JwtBearer` library to handle access tokens in the JWT format. Then you added support for authorization using the Auth0 settings taken from your Auth0 dashboard. Finally, you tested unauthorized and authorized requests to verify that everything worked as expected.

The full code of the ASP.NET Core Web API project is available in the `Auth0WebAPI` folder of **this GitHub repository**.

auth0.com | © Auth0 2022

Chapter 5 - Single-Page Applications

auth0.com | © Auth0 2022

Single-Page Applications, also known as SPAs, are applications that perform most of their user interface logic in a web browser and call APIs for backend services. They are usually built with JavaScript, which may leverage frameworks such as Angular, React, Vue, etc.

While JavaScript is the most well-known and widely used programming language running in a browser, browsers have recently started supporting **WebAssembly**, often abbreviated as WASM. WebAssembly is a binary standard instruction format **supported by the major browser engines**. It brings efficiency, speed, and compactness to your browser-based applications.

The .NET platform allows you to create Single-Page Applications without using JavaScript. Thanks to the Blazor WebAssembly framework, you can rely on your C# skills and your platform knowledge to create applications that will run within a browser. Using this framework, your application is compiled into WebAssembly and runs in browsers as an SPA.

In this chapter, you will learn how to integrate a Blazor WebAssembly application with Auth0 to add authentication and authorization services.

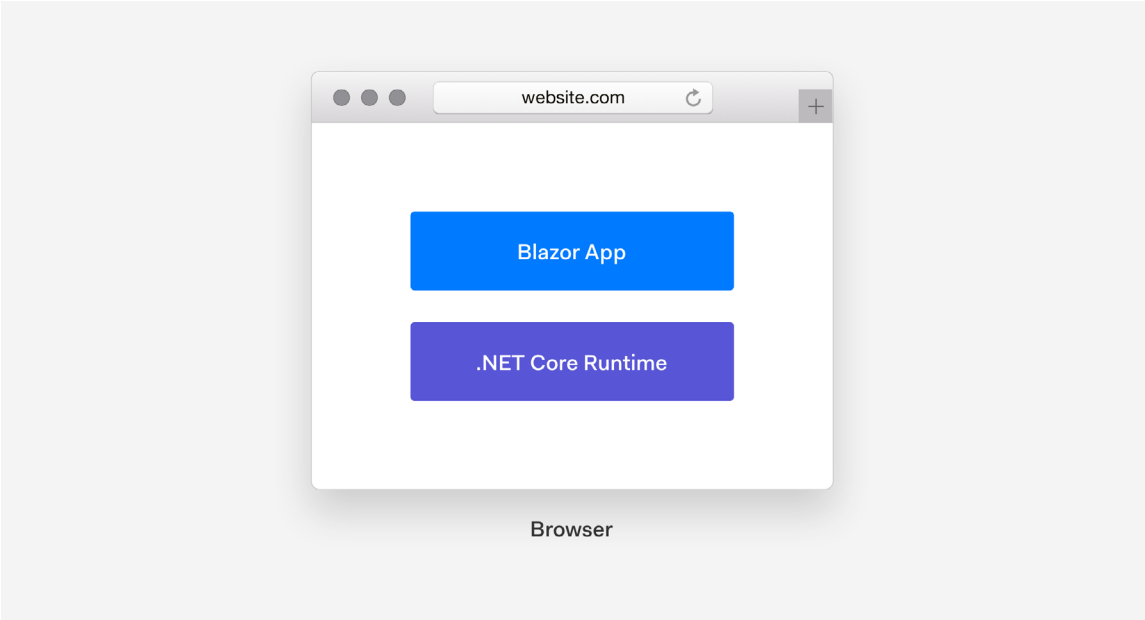
auth0.com | © Auth0 2022

Blazor WebAssembly Applications

In **Chapter 3**, you met the Blazor framework and used it to build regular web applications through its server hosting model. Now you are going to explore the same Blazor framework to build Single-Page Applications compiled into WebAssembly.

The WebAssembly hosting model

The Blazor WebAssembly hosting model, also known as *Blazor WASM*, lets your application run entirely on the user's browser. The full code of the application, including its dependencies and the .NET runtime, is compiled into **WebAssembly**, downloaded by the user's browser, and locally executed. The following picture describes the hosting model of Blazor WebAssembly:



The benefits provided by the Blazor WebAssembly hosting model are similar to those provided by Single-Page Applications. After the download, the application is independent of the server, apart from the needed interactions. Also, you don't need an ASP.NET Core Web server to host

auth0.com | © Auth0 2022

your application. You can use any Web server, since the result of the WebAssembly compilation is just a set of static files.

Depending on the structure of your project, you have two options for creating your Blazor WebAssembly application:

- You can have just the client-side application that will call an existing Web API.
- You can have both the client-side application and the Web API application. In this case, the Web API application also serves the Blazor WebAssembly app to the browsers. This option is called *ASP.NET Core hosted*.

The sample application

The sample project you will use in this section is built starting from the standard .NET template. You will create it using the *ASP.NET Core hosted* option. This means that you will have the client-side application, which will be responsible for showing the UI and managing user interactions, and the Web API application, which will provide the backend services.

Create your application by running the following command in a terminal window:

```
dotnet new blazorwasm -o Auth0BlazorWasm --hosted
```

> **Note:** If you want to create only the client-side application, you have to omit the `--hosted` flag in the previous command.

Let's run the application to make sure that everything works as expected. Go to the `Auth0BlazorWasm` folder and run the following command:

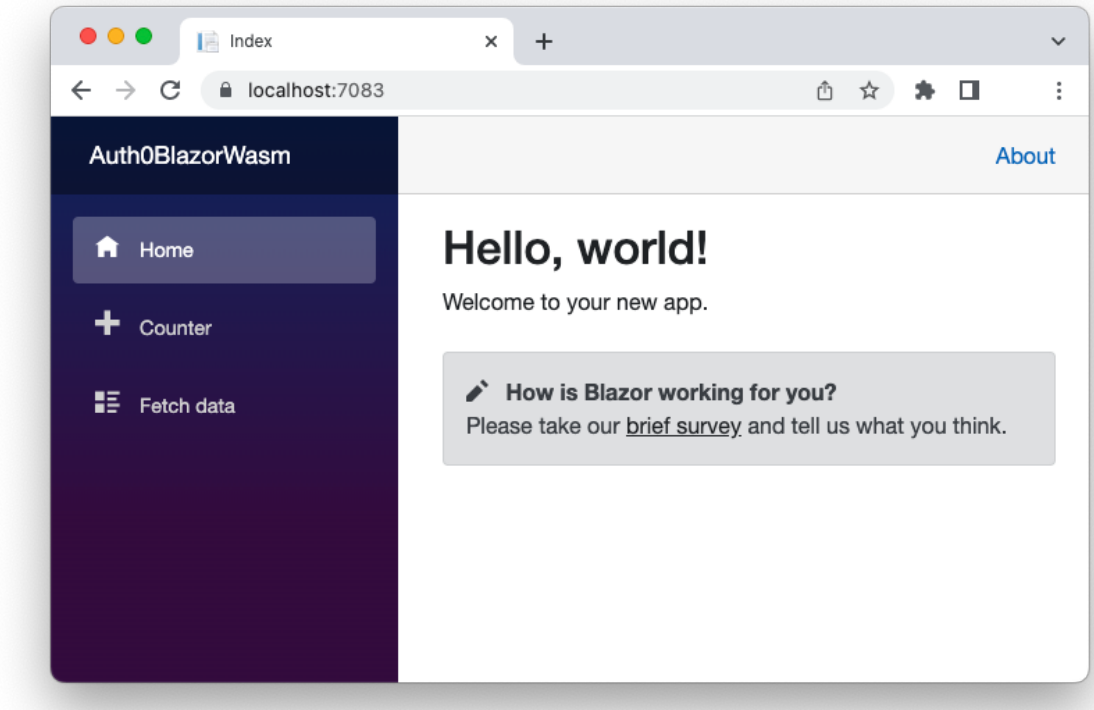
auth0.com | © Auth0 2022

```
dotnet run --project Server
```

> If you are using a Mac, you may be affected by a known issue when running an ASP.NET Core application through the .NET CLI. Please, read **this note** to learn about a workaround and apply it to the `Server/Auth0BlazorWasm.Server.csproj` file.

Take a look at your terminal window to get the address your application is listening to. It is in the form `https://localhost:<YOUR_PORT_NUMBER>`. In the examples shown in this section, the application's address is `https://localhost:7083`. Replace this address with yours wherever it's needed.

Pointing your browser to your application's address, you should see the following page:



The application built starting from the `blazorwasm` template has the same functionality as **the Blazor Server application**. It provides three pages accessible from the left-side menu:

auth0.com | © Auth0 2022

- The home page that you see in the picture above.
- The *Counter* page, which implements a simple counter incremented by clicking a button. The code of this page demonstrates how to execute code on the client side of the application without involving the server.
- The *Fetch data* page, which shows some fictitious weather forecast data. Its code demonstrates how to implement interaction between the client and server sides of the application.

Even if the look and feel of this application is basically the same as the Blazor Server implementation, the application architecture is quite different. In this case, you have a client compiled into WebAssembly and running in your browser, while the server is running in the built-in Web server. In addition, with this architecture, the client and the server interact with classic HTTP requests instead of SignalR. You can check this by analyzing the network traffic with your browser's developer tools.

If you take a look at the `Auth0BlazorWasm` folder, you will find the folder structure shown below:

```
QuizManagerClientHosted
|  Auth0BlazorWasm.sln
├── Client
├── Server
└── Shared
```

Each of these folders contains a .NET project. While the `Client` and `Server` folders are straightforward, you may wonder what the `Shared` folder contains. It contains a class library project with the code shared by the client-side and server-side applications. In this case, it contains the weather forecast data model.

auth0.com | © Auth0 2022

Register with Auth0

Let's start integrating this Blazor WebAssembly application with Auth0 by registering it. After accessing your Auth0 dashboard, go to the **Applications section**, and follow these steps:

1. Click the *Create Application* button.
2. Provide a friendly name for your application (for example, *Blazor WASM Client*) and select *Single Page Web Applications* as the application type.
3. Finally, click the *Create* button.

After you register the application, go to the *Settings* tab and take note of your Auth0 domain and client ID. Then, assign the value `https://localhost:<YOUR_PORT_NUMBER>/authentication/login-callback` to the *Allowed Callback URLs* field and the value `https://localhost:<YOUR_PORT_NUMBER>` to the *Allowed Logout URLs* field. Replace the`<YOUR_PORT_NUMBER>` placeholder with the actual port number assigned to your application.

Click the *Save Changes* button to apply them.

Now, go back to your project, go to the `Client/wwwroot` folder and create an `appsettings.json` file with the following content:

```
{
  "Auth0": {
    "Authority": "https://YOUR_DOMAIN",
    "ClientId": "YOUR_CLIENT_ID"
  }
}
```

Replace the placeholders `YOUR_DOMAIN` and `YOUR_CLIENT_ID` with the respective values taken from the Auth0 dashboard.

auth0.com | © Auth0 2022

Add authentication

To add Auth0 authentication to the project, let's apply the changes described in the following steps.

Install the authentication package

Add the authentication package to the Blazor client project by running the following command in the `Client` folder:

```
dotnet add package Microsoft.AspNetCore.Components.WebAssembly.Authentication
```

This package enables **OpenID Connect** support for your client application so that it can authenticate users with Auth0.

Set up authentication

In the `Client` folder, edit the `Program.cs` file by changing its content as follows:

```
// Client/Program.cs

using Microsoft.AspNetCore.Components.Web;
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;
using Auth0BlazorWasm.Client;

var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");
builder.RootComponents.Add<HeadOutlet>("head::after");

builder.Services.AddScoped(sp => new HttpClient { BaseAddress = new Uri(builder.
HostEnvironment.BaseAddress) });
```

auth0.com | © Auth0 2022

```
// 👇 new code
builder.Services.AddOidcAuthentication(options =>
{
  builder.Configuration.Bind("Auth0", options.ProviderOptions);
  options.ProviderOptions.ResponseType = "code";
});
// 👆 new code

await builder.Build().RunAsync();
```

You add the call to `AddOidcAuthentication()` with specific options. In particular, you specify using the parameters from the `Auth0` section of the `appsettings.json` configuration file. Also, you specify the type of **authentication and authorization flow** you want to use. In this specific case, the **Authorization Code flow** is recommended.

To complete the implementation of authentication support in your client application, open the `indexhtml` file under the `Client/wwwroot` folder and add the reference to the `AuthenticationService.js` script as shown below:

```
<!-- Client/wwwroot/index.html -->

<!DOCTYPE html>
<html>

  <!-- existing markup -->

  <body>

    <!-- existing markup -->
```

© Auth0 2022 | auth0.com

```
    <script src="_framework/blazor.webassembly.js"></script>
    <!-- 👇 new addition -->
    <script src="_content/Microsoft.AspNetCore.Components.WebAssembly.
    Authentication/AuthenticationService.js"></script>
    <!-- 👆 new addition -->
  </body>
</html>
```

This script is responsible for performing the authentication operations on the WebAssembly client side.

Implement login

At this point, you prepared the infrastructure for your Blazor app to support authentication. Now you need to make some changes to the UI to let users log in to it.

The first step is to enable support for the authorization **Razor components**. So, open the `_Imports.razor` file in the `Client` folder and add a reference to the `Microsoft.AspNetCore.Components.Authorization` and `Microsoft.AspNetCore.Authorization` namespaces. The content of that file will look as follows:

```
@* Client/_Imports.razor *@

@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization  // 👈 new addition
@using Microsoft.AspNetCore.Authorization             // 👈 new addition
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
```

© Auth0 2022 | auth0.com

```
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using QuizManagerClientHosted.Client
@using QuizManagerClientHosted.Client.Shared
```

Then, open the App.razor file in the same folder and replace its content with the following:

```
<!-- Client/App.razor -->

<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(App).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData" DefaultLayout=
            "@typeof(MainLayout)">
                <Authorizing>
                    <p>Determining session state, please wait...</p>
                </Authorizing>
                <NotAuthorized>
                    <h1>Sorry</h1>
                    <p>You're not authorized to reach this page. You need to log
                    in.</p>
                </NotAuthorized>
            </AuthorizeRouteView>
            <FocusOnNavigate RouteData="@routeData" Selector="h1" />
        </Found>
        <NotFound>
            <PageTitle>Not found</PageTitle>
            <LayoutView Layout="@typeof(MainLayout)">
                <p role="alert">Sorry, there's nothing at this address.</p>
```

auth0.com | © Auth0 2022

```
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

You used the `AuthorizeRouteView` Blazor component to customize the content according to the user's authentication status. The `CascadingAuthenticationState` component will propagate the current authentication state to the inner components so they can work on it consistently.

The next step is to create a new Razor component that allows the user to log in and see their name when authenticated. So, create a new file named `AccessControl.razor` in the `Client/Shared` folder with the following content:

```
@* Client/Shared/AccessControl.razor *@

@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

@inject NavigationManager Navigation
@inject SignOutSessionStateManager SignOutManager

<AuthorizeView>
    <Authorized>
        Hello, @context.User.Identity.Name!
    </Authorized>
    <NotAuthorized>
        <a href="authentication/login">Log in</a>
    </NotAuthorized>
</AuthorizeView>
```

auth0.com | © Auth0 2022

The component uses the `AuthorizeView` component to show different content according to the user's authentication status. Basically, it shows the *Log in* link when the user is not authenticated. It shows the name of the user when they are authenticated.

Now, open the `MainLayout.razor` file in the `Shared` folder and add the `AccessControl` component just before the *About* link. The final code should look like the following:

```
@* Client/Shared/MainLayout.razor *@

@inherits LayoutComponentBase

<div class="page">
    <div class="sidebar">
        <NavMenu />
    </div>

    <main>
        <div class="top-row px-4">
            <AccessControl />    // new code
            <a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>
        </div>

        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

When you registered your Blazor app with Auth0, you specified an allowed URL for login callback. To manage this URL, you need to implement a page responsible for handling it. To do this, create a new `Authentication.razor` file in the `Client/Pages` folder with the following code:

auth0.com | © Auth0 2022

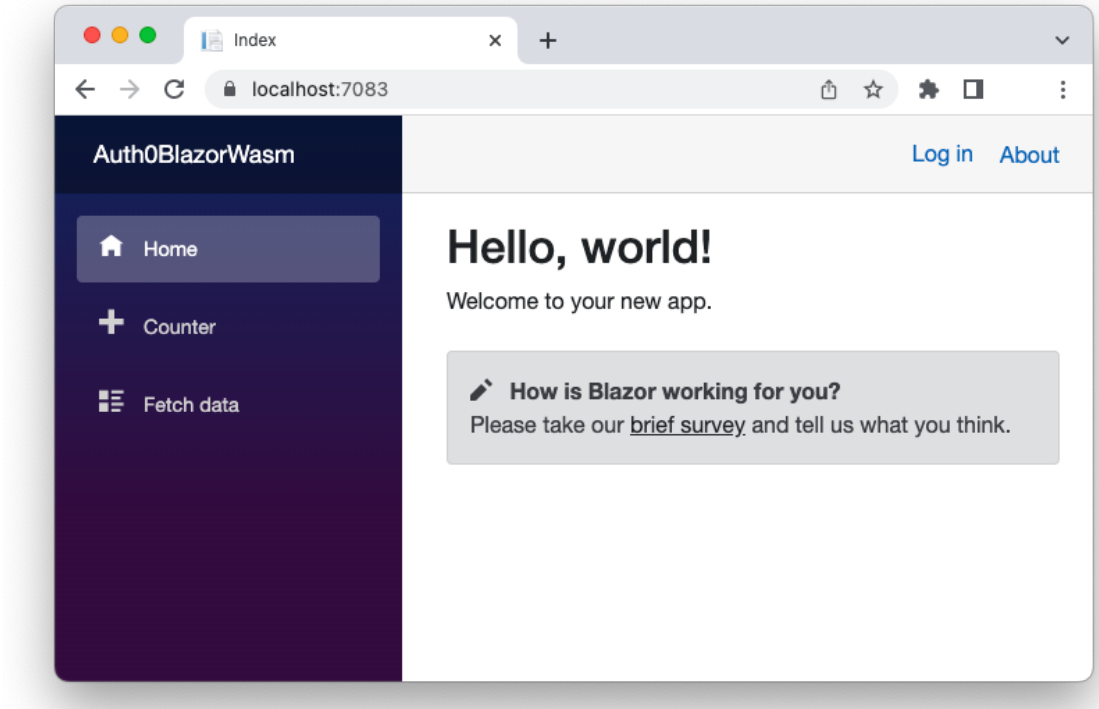
```
@* Client/Shared/Authentication.razor *@

@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Configuration

@inject NavigationManager Navigation
@inject IConfiguration Configuration

<RemoteAuthenticatorView Action="@Action">
</RemoteAuthenticatorView>

@code{
    [Parameter] public string Action { get; set; }
}
```

As you can see, this component implements a page containing the `RemoteAuthenticatorView` component. This component manages the user's authentication status and interacts with the authorization server on the Auth0 side.

Test login

At this point, you can stop your Blazor app, if it is still running, and restart it to test the authentication integration. Now its home page looks like the following picture:

auth0.com | © Auth0 2022



You can see the *Log in* link next to the *About* link. By clicking on it, the **Auth0 Universal Login page** is shown, and the authentication process takes place.

Protect private pages

So far, the pages shown when you click the *Counter* and *Fetch data* menu items are not protected from unauthorized access. To fix this issue, open the `Index.razor` file in the `Pages` folder and add the `Authorize` attribute as shown in the following code snippet:

```
@* Client/Pages/Index.razor *@

@page "/"
@attribute [Authorize]  // 👈 new code

<PageTitle>Index</PageTitle>

<h1>Hello, world!</h1>
```

auth0.com | © Auth0 2022

```
Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />
```

Add the same attribute to the `Counter.razor` component as well:

```
@* Client/Pages/Counter.razor *@

@page "/counter"
@attribute [Authorize]  // 👈 new code

<PageTitle>Counter</PageTitle>

@* ...existing code... *@
```

And finally, add that attribute to the `FetchData.razor` component:

```
@* Client/Pages/FetchData.razor *@

@page "/fetchdata"
@attribute [Authorize]  // 👈 new code

@using Auth0BlazorWasm.Shared
@inject HttpClient Http

<PageTitle>Weather forecast</PageTitle>

@* ...existing code... *@
```

These changes ensure that authorized users can access those pages.

© Auth0 2022 | auth0.com

Implement logout

To allow users to log out of your application, open the `Authentication.razor` page in the `Client/Pages` folder and change its content as follows:

```
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Configuration

@inject NavigationManager Navigation
@inject IConfiguration Configuration

<RemoteAuthenticatorView Action="@Action">
    // 👇 new code
    <LogOut>
        @{
            var authority = (string)Configuration["Auth0:Authority"];
            var clientId = (string)Configuration["Auth0:ClientId"];

             Navigation.NavigateTo($"{authority}/v2/logout?client_id={clientId}");
        }
    </LogOut>
    // 👆 new code
</RemoteAuthenticatorView>

@code{
    [Parameter] public string Action { get; set; }
}
```

While the login interaction doesn't require any specific code, you need to manage the logout transaction. In fact, by design Blazor clears your authentication state on the client side but doesn't disconnect you from Auth0. To close your session on the Auth0 side, you need to explicitly call the logout endpoint, as shown in the code above.

auth0.com | © Auth0 2022

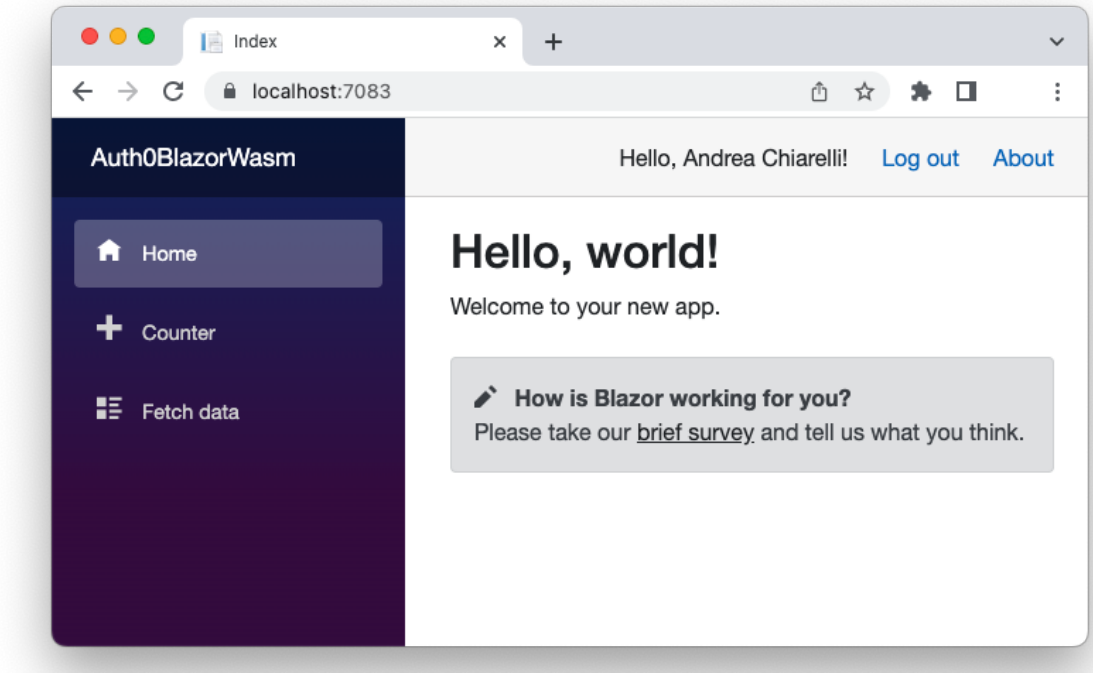
Now, let's modify the UI to enable the user to log out of the application. Open the `AccessControl.razor` file in the `Client/Shared` folder and add the code shown below:

```
@* Client/Shared/AccessControl.razor *@

@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

@inject NavigationManager Navigation
@inject SignOutSessionStateManager SignOutManager

<AuthorizeView>
    <Authorized>
        Hello, @context.User.Identity.Name!
        // 👇 new code
        <a href="#" @onclick="BeginSignOut">Log out</a>
        // 👆 new code
    </Authorized>
    <NotAuthorized>
        <a href="authentication/login">Log in</a>
    </NotAuthorized>
</AuthorizeView>

// 👇 new code
@code{
    private async Task BeginSignOut(MouseEventArgs args)
    {
        await SignOutManager.SetSignOutState();
        Navigation.NavigateTo("authentication/logout");
    }
}
// 👆 new code
```

© Auth0 2022 | auth0.com

The new code makes the *Log out* link visible when the user is logged in. It also defines the `BeginSignOut()` method, which initiates the logout process and calls the `authentication/logout` endpoint.

Now, when your users log in to your application, they will see the following screen:



They can log out of your application by clicking the *Log out* link.

> **Note:** At the time of writing, the logout function seems to be unstable due to an apparent Blazor problem. Check out **this issue in the Blazor project's repository to learn more**.

Call the API

The data shown on the *Fetch data* page is loaded from the `/weatherforecast` endpoint implemented in the server project. This API is not protected, so any client could access it. In fact, the Blazor WASM client can access it without any problem. However, in a production-ready

auth0.com | © Auth0 2022

scenario, you need to protect the API to prevent unauthorized access. Although the API security implementation is out of the scope of this tutorial, you need to make a few changes to the API in the server project to secure it.

Check out **Chapter 4** to learn more about protecting ASP.NET Core Web API applications.

Protect the API

To start to protect the API with Auth0, you need to register it first. Point your browser to the Auth0 Dashboard, move to the **API section**, and follow these steps:

1. Click the *Create API* button.
2. Provide a friendly name for your API (for example, *Weather API*) and a unique identifier (also known as *audience*) in the URL format (for example, `https://weather-api.com`).
3. Leave the signing algorithm to RS256 and click the *Create* button.

This way, Auth0 is aware of your Web API and will allow you to control access.

In the server project under the `Server` folder, open the `appsettings.json` and modify its content as follows:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
```

auth0.com | © Auth0 2022

```
  "Auth0": {
    "Domain": "YOUR_DOMAIN",
    "Audience": "YOUR_API_IDENTIFIER"
  }
}
```

Replace the `YOUR_DOMAIN` placeholder with the Auth0 domain value you used for the Blazor WASM client. Also, replace the `YOUR_API_IDENTIFIER` placeholder with the unique identifier you defined for your API in the Auth0 Dashboard: it should be `https://weather-api.com` if you kept the suggested value.

Still in the `Server` folder, run the following command to install the library that will handle the authorization process:

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Then, open the `Program.cs` file and apply the changes shown below:

```
// Server/Startup.cs

// ... exisiting code ...
using Microsoft.AspNetCore.Authentication.JwtBearer;
// 👆 new code

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
// 👇 new code
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, c =>
    {
        c.Authority = $"https://{builder.Configuration["Auth0:Domain"]}";
```

auth0.com | © Auth0 2022

```
        c.TokenValidationParameters = new Microsoft.IdentityModel.Tokens.
        TokenValidationParameters
        {
            ValidAudience = builder.Configuration["Auth0:Audience"],
            ValidIssuer = $"https://{builder.Configuration["Auth0:Domain"]}"
        };
    });
// 👆 new code

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();

// ... exisiting code ...

app.UseRouting();

// 👇 new code
app.UseAuthentication();
app.UseAuthorization();
// 👆 new code

app.MapRazorPages();

// ... exisiting code ...
```

You added the reference to the `Microsoft.AspNetCore.Authentication.JwtBearer` namespace and added the statements that configure the server to handle the authorization process through Auth0. Finally, you configured the middleware to process authentication and authorization.

© Auth0 2022 | auth0.com

Now, open the `WeatherForecastController.cs` file in the `Server/Controllers` folder and apply the following changes:

```
using Microsoft.AspNetCore.Mvc;
using Auth0BlazorWasm.Shared;
using Microsoft.AspNetCore.Authorization;  // 👈 new addition

namespace Auth0BlazorWasm.Server.Controllers;

[ApiController]
[Route("[controller]")]
[Authorize]             // 👈 new addition
public class WeatherForecastController : ControllerBase
{
  // ... existing code ...
}
```

You added the reference to the `Microsoft.AspNetCore.Authorization` namespace and decorated the `WeatherForecastController` class with the `Authorize` attribute.

Your API is now protected from unauthorized access.

Call the protected API

To enable your Blazor WASM application to access the protected API, you need to get an access token from Auth0 and provide it along with your API call. You might think to write some code that attaches this token when you make an HTTP request to the server. However, you can centralize the access token attachment to your API calls in a straightforward way.

Start by going to the `Client` folder and installing the `Microsoft.Extensions.Http` package with the following command:

auth0.com | © Auth0 2022

```
dotnet add package Microsoft.Extensions.Http
```

This package allows you to create named HTTP clients and customize their behavior. In your case, you will create an HTTP client that automatically attaches an access token to each HTTP request.

Open the `Program.cs` file in the `Client` folder and add a reference to the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` as shown below:

```
// Client/Program.cs

using Microsoft.AspNetCore.Components.Web;
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;
using Auth0BlazorWasm.Client;
// 👇 new addition
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

// ... existing code ...
```

In the same file, apply the changes pointed out in the following code snippet:

```
// Client/Program.cs

// ... existing code ...

builder.RootComponents.Add<HeadOutlet>("head::after");

// 👇 old code
//builder.Services.AddScoped(sp => new HttpClient { BaseAddress = new Uri(builder.
HostEnvironment.BaseAddress) });
```

auth0.com | © Auth0 2022

```
// 👇 new code
builder.Services.AddHttpClient("ServerAPI",
      client => client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress))
    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();

builder.Services.AddScoped(sp => sp.GetRequiredService<IHttpClientFactory>()
  .CreateClient("ServerAPI"));
// ☝️ new code

// ... existing code ...
```

You replaced the existing line of code that created an HTTP client with two lines of code. The `AddHttpClient()` method defines a named `HttpClient` instance (`ServerAPI`) with the current server's address as the base address to use when requesting a resource. Also, the `BaseAddressAuthorizationMessageHandler` class is added to the `HttpClient` instance as the HTTP message handler. This class is provided by the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` namespace and is responsible for attaching the access token to any HTTP request to the application's base URI.

The actual `HttpClient` instance is created by the `CreateClient()` method of the `IHttpClientFactory` service implementation.

Now, open the `appsettings.json` file in the `Client/wwwroot` folder and add the `Audience` element as shown below:

```
{
  "Auth0": {
    "Authority": "https://YOUR_DOMAIN",
    "ClientId": "YOUR_CLIENT_ID",
    "Audience": "YOUR_API_IDENTIFIER"
  }
}
```

auth0.com | © Auth0 2022

Replace the `YOUR_API_IDENTIFIER` placeholder with the unique identifier you defined for your API in the Auth0 dashboard (e.g., `https://weather-api.com`).

Now, back in the `Client/Program.cs` file, apply the change highlighted below:

```
// Client/Program.cs

// ... existing code ...

builder.Services.AddOidcAuthentication(options =>
{
  builder.Configuration.Bind("Auth0", options.ProviderOptions);
  options.ProviderOptions.ResponseType = "code";
  // 👇 new code
  options.ProviderOptions.AdditionalProviderParameters.Add("audience", builder.
  Configuration["Auth0:Audience"]);
});

await builder.Build().RunAsync();
```

You added an additional `audience` parameter to let Auth0 know you want to call the API identified by the `Audience` setting value.

After this global configuration, you can call the `weatherforecast` endpoint of your Web API. So, open the `FetchData.razor` file in the `Client/Pages` folder and change its content as follows:

```
@* Client/Pages/QuizViewer.razor *@

@page "/fetchdata"
@attribute [Authorize]
```

auth0.com | © Auth0 2022

```
@using Auth0BlazorWasm.Shared
// 👇 new code
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
// 👆 new code
@inject HttpClient Http

// ... existing code ...

@code {

    // ... existing code ...

    protected override async Task OnInitializedAsync()
    {
        // 👇 changed code
        try
        {
            forecasts = await Http.GetFromJsonAsync<WeatherForecast[]>
            ("weatherforecast");
        }
        catch (AccessTokenNotAvailableException exception)
        {
            exception.Redirect();
        }
        // 👆 changed code
    }

  // ... existing code ...
}
```

You imported the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` namespace. Then, you simply arranged the `OnInitializedAsync()` method by wrapping it with a *try-catch* statement.

auth0.com | © Auth0 2022

After applying these changes, restart your application, log in, and try to go to the *Fetch data* page. This time you should be able to access your protected API and show the weather forecast data.

Summary

This section guided you in creating and securing a Blazor WebAssembly application by using Auth0. You learned how to build a simple Blazor WebAssembly application and some Razor components. You went through the process of registering your application with Auth0 and enabling it to support authentication. Finally, you protected the API hosted by the server side of your application and called that API, passing the access token.

The full source code of the application secured in this section can be downloaded from the `Auth0BlazorWasm` folder of **this GitHub repository**).

auth0.com | © Auth0 2022

Conclusion

By the end of this book, you should have a good understanding of Auth0 and a working knowledge of how to integrate your .NET applications with its authentication and authorization services.

Along the way, you've explored the fundamental concepts of modern identity and learned how to move through the tools Auth0 provides to register and configure your application.

You then examined how to integrate various types of .NET applications with Auth0: from regular web applications to APIs to Single-Page Applications. For each type of application you put your hands in the code and experienced firsthand how you can add authentication and authorization.

You analyzed several types of .NET applications, but they do not cover all the possibilities that the development platform provides. For example, native applications and gRPC-based microservices were not included.

Also, on the Auth0 side, you have not explored all the capabilities the platform provides for you to better manage the security of your application. For example, you didn't receive guidance on how to manage permissions and roles or how to customize the Universal Login page and the user authentication or registration flows via Actions, etc.

This might be a good reason to continue the objective of this book with a second edition. In the meantime, check out the **official Auth0 documentation** and follow the **technical blog**.

auth0.com | © Auth0 2022

About Auth0

The Auth0 Identity Platform, a product unit within Okta, takes a modern approach to identity and enables organizations to provide secure access to any application, for any user. Auth0 is a highly customizable platform that is as simple as development teams want and as flexible as they need. Safeguarding billions of login transactions each month, Auth0 delivers convenience, privacy, and security so customers can focus on innovation.

For more information, visit **https://auth0.com**

auth0.com | © Auth0 2022

Copyright © 2022 by Auth0® Inc.
All rights reserved. This eBook or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations.