**How to Setup and Configure ASP.NET Core Identity**

Last Updated: September 8, 2022

**Page Contents**

* [Create the Example Project](https://www.yogihosting.com/aspnet-core-identity-setup/#project)
* [Installing Identity Packages](https://www.yogihosting.com/aspnet-core-identity-setup/#packages)
* [Configuring the Project](https://www.yogihosting.com/aspnet-core-identity-setup/#configuration)
* [Setting up ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-setup/#setup)
  + [User Class](https://www.yogihosting.com/aspnet-core-identity-setup/#user-class)
  + [Database Context](https://www.yogihosting.com/aspnet-core-identity-setup/#dbcontext)
  + [Database Connection String](https://www.yogihosting.com/aspnet-core-identity-setup/#database)
* [Set up ASP.NET Core Identity as a Service](https://www.yogihosting.com/aspnet-core-identity-setup/#service)
* [Creating the Identity Database using the EF Core Migration Commands](https://www.yogihosting.com/aspnet-core-identity-setup/#migrations)
* [ASP.NET Core Identity Database](https://www.yogihosting.com/aspnet-core-identity-setup/#db)
* [Download Source Codes](https://www.yogihosting.com/aspnet-core-identity-setup/#download)

**ASP.NET Core Identity** is a Toolkit and an API with which you can create **Authorization and Authentication** features in your application. Users can create an account and login with a user name and password. This also includes **Roles and Roles Management**. ASP.NET Core Identity uses a SQL Server Database to store user names, passwords, roles, and profile data.

This tutorial requires the SQL Server LocalDB feature to be installed for Visual Studio. You can add this feature by running the Visual Studio installer and installing the SQL Server Express LocalDB option from the Individual Components section.

You can also directly download this from the Microsoft website – [Link](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-express-localdb?view=sql-server-ver16)

This chapter is a part of the ASP.NET Core Identity lessons

* 1. *How to Setup and Configure ASP.NET Core Identity*
* 2. [How to Create, Read, Update & Delete users in ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-create-read-update-delete-users/)
* 3. [Username, Email & Password Policy in ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-username-email-password-policy/)
* 4. [How to do Authentication of Users in ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-authentication/)
* 5. [How to work with Roles in ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-roles/)
* 6. [How to add Custom User Properties in ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-custom-user-properties/)
* 7. [How to work with Claims in ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-claims/)
* 8. [How to work with Policies in ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-policies/)
* 9. [How to integrate Google login feature in ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-login-with-google/)
* 10. [Two-Factor Authentication in ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-two-factor-authentication/)
* 11. [How to perform Email Confirmation of Users in ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-email-confirmation/)
* 12. [Creating Password Reset feature in ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-password-reset/)
* 13. [User Lockout in ASP.NET Core Identity](https://www.yogihosting.com/aspnet-core-identity-user-lockout/)

Create the Example Project

Best way for understanding **ASP.NET Core Identity** is through an example. So we create a new ASP.NET Core MVC project and name it Identity. Next, we will install the necessary packages and then configure Identity in this example project.

Installing Identity Packages

**ASP.NET Core Identity** needs 4 packages to be installed in your application. These packages are:

* 1. Microsoft.AspNetCore.Identity
* 2. Microsoft.AspNetCore.Identity.EntityFrameworkCore
* 3. Microsoft.EntityFrameworkCore.Design
* 4. Microsoft.EntityFrameworkCore.SqlServer

Install them from NuGet, first go to Tools ➤ NuGet Package Manager ➤ Manage NuGet Packages for Solutions in your Visual Studio. Then search for these packages and click install. I have provided with a screenshot of these packages in the below image.

Note that install only the supported versions of the packages based on the .NET version you are using.

Configuring the Project

Go to the Startup.cs class and add Authentication and Authorization [Middlewares](https://www.yogihosting.com/aspnet-core-configurations/" \l "middleware" \t "_blank) that will be needed for the project. So add the below 2 lines of code below app.UseRouting().

app.UseAuthentication();

app.UseAuthorization();

Setting up ASP.NET Core Identity

The process for setting up **ASP.NET Core Identity** involves creating new model classes, configuration changes, controllers and actions to support **authentication and authorization operations**.

User Class

User class defines the Users for the application. These users are stored in the database. The User class is derived from IdentityUser class of Microsoft.AspNetCore.Identity namespace.

So, create AppUser.cs class inside the “Models” folder with the following code:

|  |  |
| --- | --- |
|  | using System;  using System.Collections.Generic;  using System.Linq;  using Microsoft.AspNetCore.Identity;    namespace Identity.Models  {      public class AppUser: IdentityUser      {      }  } |

Notice, the AppUser.cs does not contain any property or method. This is because the IdentityUser.cs class provides it with some of the properties like the **user name, e-mail, phone number, password hash, role memberships** and so on.

If you want more properties that are not defined in the IdentityUser class then you can certainly add them to your User Class. I will take this in the later on sections.

The IdentityUser class properties are defined below:

| **Name** | **Description** |
| --- | --- |
| Id | It contains the Unique Id of the user. |
| UserName | It contains the user’s username. |
| Claims | This property returns all the claims for the user. |
| Email | It contains the email for the user. |
| PasswordHash | It contains the hash form of the user password. |
| Roles | It returns all the roles for the user. |
| PhoneNumber | It returns the phone number for the user. |
| SecurityStamp | It contains a value that is changed whenever the user identity is altered. For example a password or phone number change, etc. |

Database Context

The **Database Context Class is derived** from the IdentityDbContext<T> where T is the User class that is AppUser in the project. The **IdentityDbContext** class is from Microsoft.AspNetCore.Identity.EntityFrameworkCore namespace and is used by Identity for communicating with the database through Entity Framework Core.

The **Database Context Class** operates on the User class. Remember we have defined the User class as AppUser.cs in the above section.

Now, create the **DB Context Class** inside the Models folder. You can do it by adding a new class and give it a name as AppIdentityDbContext.cs. Next add the following code to it:

|  |  |
| --- | --- |
|  | using Microsoft.AspNetCore.Identity.EntityFrameworkCore;  using Microsoft.EntityFrameworkCore;    namespace Identity.Models  {      public class AppIdentityDbContext : IdentityDbContext<AppUser>      {          public AppIdentityDbContext(DbContextOptions<AppIdentityDbContext> options): base(options) { }      }  } |

Notice, I haven’t added any property or method to this class, this is because this empty class is inheriting from IdentityDbContext<AppUser> and so it is getting all it’s methods and properties.

Database Connection String

The **ASP.NET Core Identity Database Connection string** contains the Database name, user name and password to connect to the Identity Database. It is stored in the appsettings.json file. This file resides on the root of the project.

Mostly this file is already created in a new project. If it is missing then you right click the project name in the Solution Explorer, and select Add ➤ New Item. Then on the list of items, select App Settings File, and click the Add button.

Add the following code to your appsettings.json file.

{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=IdentityDB;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

}

In the Connection string, the Server name specifies the SQL Server LocalDB, Database name specifies the name of the database which I have kept as IdentityDB. You can name it anything which you want.

The Trusted\_Connection is set to ‘True’ so the project will use windows authentication to connect to this database and there is no need for providing the database username and password.

Multiple Active Result Sets (MARS) is a feature that works with SQL Server to allow the execution of multiple batches on a single connection. This gives faster execution of SQL Operations. I have set it to true – MultipleActiveResultSets=true.

Now, I have to update the Startup.cs file so that the application can read the connection string from the appsettings.json file.

The changes that are made to the Startup.cs class are highlighted (see below).

|  |
| --- |
| using Microsoft.AspNetCore.Builder;  using Microsoft.AspNetCore.Hosting;  using Microsoft.Extensions.DependencyInjection;  using Microsoft.Extensions.Hosting;  using Identity.Models;  using Microsoft.Extensions.Configuration;  using Microsoft.EntityFrameworkCore;    namespace Identity  {      public class Startup      {          public Startup(IConfiguration configuration) => Configuration = configuration;          public IConfiguration Configuration { get; }            public void ConfigureServices(IServiceCollection services)          {              services.AddDbContext<AppIdentityDbContext>(options => options.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"]));              services.AddControllersWithViews();          }            public void Configure(IApplicationBuilder app, IWebHostEnvironment env)          {              //...          }      }  } |

First you have to add a constructor to receive an object of type IConfiguration. It will be provided by the [Dependency Injection feature of ASP.NET Core](https://www.yogihosting.com/aspnet-core-dependency-injection/). The added code is:

**public Startup(IConfiguration configuration) => Configuration = configuration;**

**public IConfiguration Configuration { get; }**

Next, inside the ConfigureServices() method I instruct the project how it has to get the connection string to the database.

I used the AddDbContext() method to apply the **DB Context Class** (that I created earlier) and specified that it will be using the **SQL Server database** whose connection string is obtained from the application’s configuration file, which in my case is appsettings.json file.

The applicable code is given below:

services.AddDbContext<AppIdentityDbContext>(options => options.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"]));

For .NET 6.0 and later

In DOT NET 6.0 and later versions the Startup class is not there, instead all these codes have to be written to Program.cs class. So below is the updated code (shown highlighted) for the .NET 6.0 and later versions.

|  |  |
| --- | --- |
|  | using Identity.Models;  using Microsoft.AspNetCore.Identity;  using Microsoft.EntityFrameworkCore;    var builder = WebApplication.CreateBuilder(args);    builder.Services.AddDbContext<AppIdentityDbContext>(options => options.UseSqlServer(builder.Configuration["ConnectionStrings:DefaultConnection"]));    // Add services to the container.  builder.Services.AddControllersWithViews();    var app = builder.Build();    //..... |

Set up ASP.NET Core Identity as a Service

Now have to perform the **ASP.NET Core Identity Configuration** on the Startup.cs class. The updated code for Startup.cs class is given below:

|  |  |
| --- | --- |
|  | using Microsoft.AspNetCore.Builder;  using Microsoft.AspNetCore.Hosting;  using Microsoft.Extensions.DependencyInjection;  using Microsoft.Extensions.Hosting;  using Identity.Models;  using Microsoft.Extensions.Configuration;  using Microsoft.EntityFrameworkCore;  using Microsoft.AspNetCore.Identity;    namespace Identity  {      public class Startup      {          public Startup(IConfiguration configuration) => Configuration = configuration;          public IConfiguration Configuration { get; }            public void ConfigureServices(IServiceCollection services)          {              services.AddDbContext<AppIdentityDbContext>(options => options.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"]));              services.AddIdentity<AppUser, IdentityRole>().AddEntityFrameworkStores<AppIdentityDbContext>().AddDefaultTokenProviders();              services.AddControllersWithViews();          }            public void Configure(IApplicationBuilder app, IWebHostEnvironment env)          {              //...          }      }  } |

Note – inside the ConfigureServices() method I have added the given code line to set up the **ASP.NET Core Identity as a Service**:

services.AddIdentity<AppUser, IdentityRole>().AddEntityFrameworkStores<AppIdentityDbContext>().AddDefaultTokenProviders();

The above code specifies that:

* The AddIdentity method’s type parameter are specified with the class used to represent users (AppUser) and the class used to represent roles (Identity Role).
* The AddEntityFrameworkStores method specifies that Identity should use EF Core and the DB Content class which in the project is AppIdentityContext class.
* The AddDefaultTokenProviders method adds the default token providers used to generate tokens for reset passwords, change email and change telephone number operations, and for two factor authentication token generation.

Recall we earlier added app.UseAuthentication() to the Startup.cs . By this on every HTTP Request, the user’s credentials will be added on a Cookie or URL. This will associate a give user with the HTTP request he or she makes to the application.

.NET 6.0 & later Versions

Below is the code for the .NET 6 and later versions.

|  |  |
| --- | --- |
|  | **using Identity.Models;**  **using Microsoft.AspNetCore.Identity;**  **using Microsoft.EntityFrameworkCore;**    **var builder = WebApplication.CreateBuilder(args);**    **builder.Services.AddDbContext<AppIdentityDbContext>(options => options.UseSqlServer(builder.Configuration["ConnectionStrings:DefaultConnection"]));**  **builder.Services.AddIdentity<AppUser, IdentityRole>().AddEntityFrameworkStores<AppIdentityDbContext>().AddDefaultTokenProviders();**    **// Add services to the container.**  **builder.Services.AddControllersWithViews();**    **var app = builder.Build();**    **//...** |

Creating the Identity Database using the EF Core Migration Commands

Now I am ready to create my Identity Database. I will use **Entity Framework Core Migration** commands to perform this task.

To run these Migration Commands I will use the Package Manager Console window. You will also need to install **.NET Core CLI** which are Entity Framework Core tools for running and managing migrations. Install it by running the following command on the Package Manager Console window:

dotnet tool install --global dotnet-ef

Next you need to go to the directory of the project file – .csproj. For this, run the dir command on the Package Manager Console window. The dir command will tell the current directory location.

When I ran the dir command it showed me my current direction location. As shown in the image given below, there are – Identity directory and Identity.sln file but no project file.

The Project file is located inside the Identity folder. So have to go inside the Identity folder by using **the cd ./Identity.**

Then I again ran the dir command which shows the Identity.csproj file, as shown in the image given below:

So now I can run my **EF Core Migration** Commands.

The first command to run is:

dotnet ef migrations add MyCommand1

The first command will take few seconds time to complete. Once completed, it will create the SQL Statements, necessary to create the database, in a few files. These files are created inside the Migrations folder.

The second command to run now is:

dotnet ef database update

This command will create your database from the files created by the first command.

Migrations with Package Manager Console (PMC) tools

You can simply skip .NET Core CLI commands and perform migrations with Package Manager Console (PMC) tools. For this you need to install the NuGet package – Microsoft.EntityFrameworkCore.Tools. Then run the following 2 commands on the Package Manager Console window.

**Add-Migration MyCommand1**

**Update-Database**

ASP.NET Core Identity Database

The Identity database is created by the migration commands. You can check your newly created database from the SQL Server Object Explorer window. See the below image which shows this newly created IdentityDB database.

Note: The SQL Server Object Explorer window can be opened from the View area given in the Visual Studio menu.

The Identity database has 8 tables in total. These tables contains the users record, roles, claims, token and login time details for these users.