Introduction to ASP.NET Core Identity

In this lesson we will learn about the need for a membership system to track the multiple aspects related to your microservices users, how ASP.NET Core Identity can help and how to leverage it in our upcoming Identity microservice.

The need for a membership system

As you know, we have already introduced the concept of a user into the PlayEconomy system. Thanks to our Inventory microservice we can both grant Catalog items and query for all the items in the user's inventory bag. However, at this point we don't know much about this user other than the user id.

If we get on the shoes of our users or players for a moment, we will realize that there are a bunch of things that we would like to know or do in the system, but we are not able to do yet. For instance, as a player, to acquire catalog items into my inventory I will need to pay for them first, however I still don't know how much gil I have available or where this is stored.

If my gil is stored somewhere in the PlayEconomy system, I would like to know how to access the system and, in fact, how do I become part of the system to start with. Then, as I register in the system, I'd like to know if my personal data is protected, especially things like my sign in credentials. Is a combination of username and password enough to enter the system or can additional layers of protection be put in place?

Also, after I sign in to the system, what can I actually do there? Can I see or modify the inventory of other players? Can I happily grant 1,000,000 gil to my own account to go and purchase items almost without limits? There should probably be some boundaries there.

As you may realize by now, we need a system to take care of all of this, specifically a membership system.

ASP.NET Core Identity

Luckily, there is a membership system already baked into the .NET platform and it's called ASP.NET Core Identity.

This membership system brings in APIs to perform all the basic user management tasks such as registering a new user, getting the list of registered users and updating or deleting existing users. This includes APIs to send an email confirmation after the user registers and to perform the password recovery process.

There are also APIs to sign in and sign out users and to manage the different roles you might want to enable in your application. For instance, in our system, you might want to separate our players, which can only perform a few limited actions, from our administrators, which can perform all sorts of management related tasks across the whole system.

A great feature of ASP.NET Core Identity is the fact that it includes prebuilt UI functionality for user registration, login, logout and personal data management. You can take advantage of these UI pieces as-is or you can also customize all or part of it as you see fit for your application.

You can also enable more advanced features like multi factor authentication or perhaps you would prefer to delegate the authentication piece to external login providers like Facebook, Twitter, Microsoft or Google. Finally, you can also store the additional claims and access tokens from these external authentication providers into ASP.NET Core Identity to offer more advanced identity related scenarios within your app.

Let's now see how ASP.NET Core Identity fits into our PlayEconomy system.

Implementing the Identity microservice

To introduce all these identity related capabilities to our system we will standup a brand new microservice called Identity. This microservice will be directly integrated with ASP.NET Core Identity to enable all the user and role management capabilities needed in our system. Let's take a quick look at the initial implementation of this new service.

ASP.NET Core Identity includes a UI library based on the Razor Pages framework. We will take advantage of these UI components to enable user registration and sign in. That way we don't have to invent the wheel around these basic UI scenarios.

We will then implement a REST API for user management in a similar way to how we implemented REST APIs in our other microservices. That way we can have operations to get the details of our users, to update details like the amount of gil assigned to a user, and also to enable user deletion.

The good thing is that this time we don't have to implement a repository or any piece of code to interact with a database. ASP.NET Core Identity includes a series of classes called "managers" that allow you to easily interact with the different components of the membership system. For instance, you can use the UserManager to perform all the usual CRUD operations involving our users, the role manager for all operations involving roles and the SignInManager to perform user sign in, sign out and a few other operations.

These identity managers interact with what we call Identity Stores, which are concrete data access implementations to interact with a specific data store provider. ASP.NET Core Identity comes with built in stores based on the Entity Framework Core object relational mapper, which by default is configured to store data in a SQL Server database.

We could certainly use this default implementation, but since we are already using MongoDB as our database server in our previous microservices, we will use a MongoDB based stores implementation instead. All the identity managers will be configured to use these MongoDB stores to persist users and roles to a new MongoDB database owned by the Identity microservice, therefore avoiding the need to bring in SQL Server into the picture.

Now, in terms of what actually needs to get implemented, we will use most of the built-in razor pages as-is, but we will need to add a few modifications to the page used for user registration, just so that we can set an initial amount of gil for each new player. Implementing the Users REST API is also all on us since that is not available by default in ASP.NET Core Identity, although the actual implementation is very straightforward since all the identity managers are already built and they are generalized in such a way that the REST API does not need to know which is the specific configured identity store.

Also, we won't need to implement the MongoDB Stores since there are multiple NuGet packages publicly available that can do this job for us, so we will just pick the one that works best for our needs and configure it in our microservice.

Let's go ahead and create our Identity microservice.