Adding Authorization

In this lesson you will learn about the need for authorization checks in your microservices and the different ways ASP.NET Core allows you to perform authorization.

The need for authorization

So far we are able to assert who our users are, that is, we can authenticate our users. However, we have not yet defined the permissions our users have across our microservices, which is what authorization is all about. To understand why this is important let's go through one of our scenarios once again.

John would like to add an item to our catalog, let' say a potion, via a web client. Let's assume the web client already acquired an access token after our Identity microservice authenticated John. The client then sends the request to create the potion to our Catalog microservice, including the access token.

Our Catalog microservice validates the token presented by our web client and confirms that the token contains claims that assert that a user authenticated and that user was John. However, Catalog now has more questions about this request. For instance, Catalog should likely only allow modifications to the items collection to Admin users, otherwise any player could, for instance, modify the prices of any item in the catalog, which is definitively a bad idea. So, is John an Admin? Catalog can't tell this yet.

Also, assuming Catalog can assert that John is an Admin, there's still the question of what kind of permissions the Web Client was granted on the Catalog to start with. For instance, we might be fine with allowing our web client to modify the catalog, but if the client is actually a native client, like a mobile app, we may want to only allow read operations, since we have not performed a proper security review of this app yet.

Only after Catalog can answer these additional questions it might allow the creation of the item in its database. And to do this we need to enable authorization across our microservices.

ASP.NET Core role-based authorization

One way to enable authorization in ASP.NET Core apps is by using roles. Role-based authorization allows limiting access to resources based on the roles the user belongs to. But what is a role? Well, really it is nothing more than a well-known claim that can be associated to a user. Our Identity microservice can keep track of the roles associated to each user and when our other microservices receive this new claim as part of the access token they can decide what kind of permissions to grant on their owned resources based on the role.

So, for instance, if the role associated to our user is "Player", our Inventory microservice might decide to only allow the GET operation that retrieves the inventory bag of a player and our Identity microservice would only allow getting the current amount of gil assigned to the authenticated player. The Catalog microservice would not allow any operations to players.

On the other hand, if the user's role is an Admin, Inventory does allow any kind of operation on any user's inventory bag, the Identity microservice allows any operation on the details of any user and Catalog also allows modifying its collection of items in any way.

So, different roles grant different kinds of permissions on each microservice.

ASP.NET Core claims-based authorization

Another way to verify permissions is via what is known as claims-based authorization. In this model access to resources can be limited based on any of the claims presented in the access token. ASP.NET Core enables this by allowing you to define policies that group a series of claims requirements that can be verified on any of the operations of the microservice REST API.

For instance, our Catalog microservice could define a Read policy and apply it to all it's GET operations. This policy would require that the value of the Role claim in the access token must be Admin and the value of the scope claim must include catalog.readaccess or catalog.fullaccess.

There could also be a Write policy where, again, the presented role must be Admin but the scope must be either catalog.writeaccess or catalog.fullaccess. You could extend these policies to require any specific value in any other claims presented in the access token.

As you can see, claims-based authorization allows you to define much more granular and complex rules than role-based authorization.

In the next lesson we will seed users and roles in our Identity microservice so we can later use the roles to perform role-based authorization across all of our microservices.