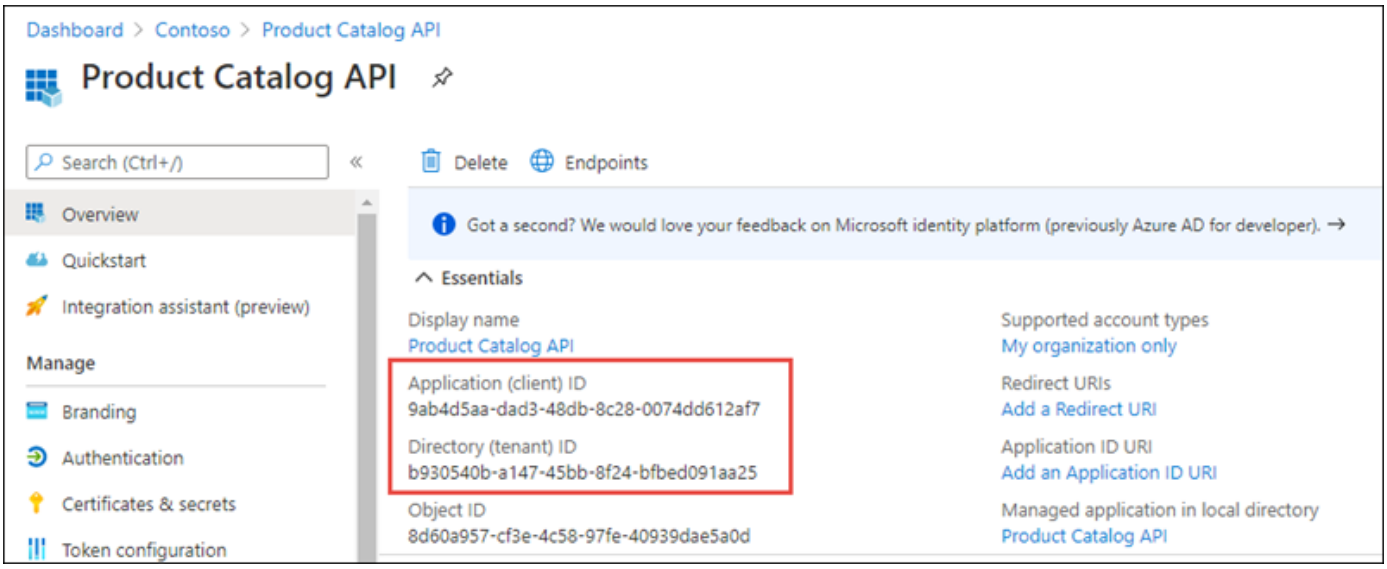
On the **Register an application** page, set the values as follows:

- **Name**: Product Catalog API
- **Supported account types**: Accounts in this organizational directory only (Single tenant)

Select **Register** to create the application.

On the **Product Catalog API** page, copy the values **Application (client) ID** and **Directory (tenant) ID**; you'll need the value of the tenant ID later in this exercise and you'll need the value of the client ID in all of the exercises in this module.



Select **Expose an API** in the left-hand navigation.

Select **Add a scope**.

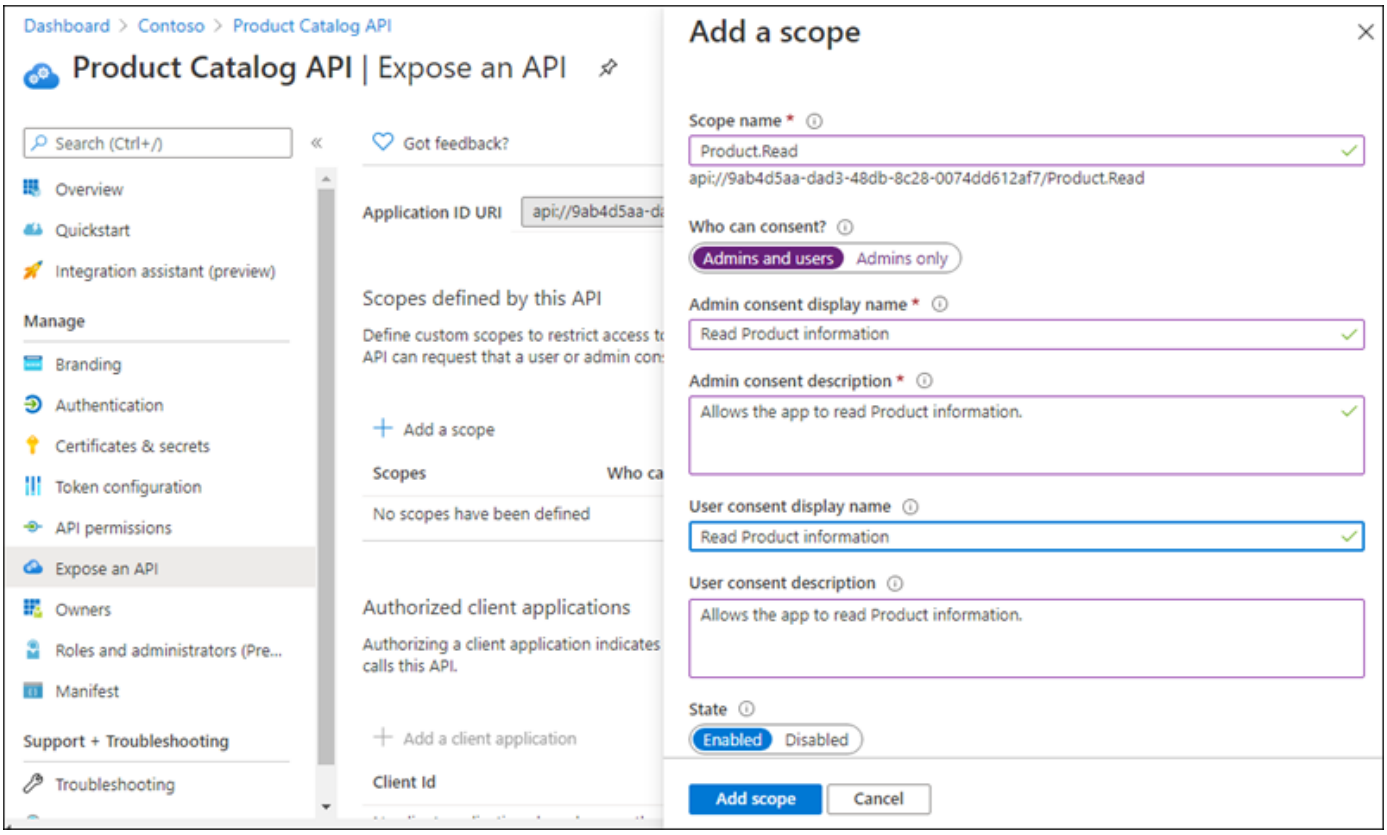
If prompted, accept the proposed application ID URI, `api://{clientId}`, by selecting **Save and Continue**.

On the **Add a scope** panel, set the values as follows:

- **Scope name**: Product.Read
- **Who can consent**: Admins and users
- **Admin consent display name**: Read Product information
- **Admin consent description**: Allows the app to read Product information.
- **User consent display name**: Read Product information
- **User consent description**: Allows the app to read Product information.
- **State**: Enabled

Select **Add scope**

Select **Add scope**.



Repeat the steps to add the following scopes, allowing **Admin and users** consent:

- **Product.Write**
- **Category.Read**

The API can require administrative consent for specific scopes as well. Create a scope requiring admin consent by specifying the values as follows on the **Add a scope** panel:

- **Scope name**: Category.Write
- **Who can consent**: Admins only
- **Admin consent display name**: Write Product Category information
- **Admin consent description**: Allows the app to write Product Category information.
- **State**: Enabled

Create a .NET Core web API application

> **Note**
>
> The instructions below assume you are using .NET 5. They were last tested using v5.0.202 of the .NET 5 SDK.

A web API application is typically a dynamic web application that is called by client applications, returning information as JSON. This example will use an Azure AD application to authenticate calls made to the application using a token provided in the Authentication header of the Http request.

Open your command prompt, navigate to a directory where you want to save your work.

Execute the following command to create a new .NET Core web API application:

Console

```
dotnet new webapi -o ProductCatalog -au singleorg
```

After creating the application, run the following commands to ensure your new project runs correctly.

Console

```
cd ProductCatalogWeb
dotnet add package Microsoft.Identity.Web
```

Open the scaffolded project folder, which is named **ProductCatalog** in **Visual Studio Code**. When a dialog box asks if you want to add required assets to the project, select **Yes**.

The scaffolded project contains a controller for weather forecasts that isn't needed. Delete the following files:

- **WeatherForecast.cs**
- **Controllers\WeatherForecastController.cs**

The web API application will run concurrently with other web applications in later modules. Each application must bind to a different TCP port. Update this web API application to use a specific port:

- Locate and open the **./.vscode/launch.json** file.
- Locate the configuration named `.NET Core Launch (web)`. Update the `env` attribute of that configuration to include the `ASPNETCORE_URLS` property:

  JSON

  ```
  "env": {
  ```

```
    "ASPNETCORE_URLS":"https://localhost:5050"
  }
```

The web API application doesn't contain any HTML pages, so there's no need to launch the browser. In the **launch.json** file, locate and remove the entire `serverReadyAction` node.

JSON

```
// "serverReadyAction": {
//   "action": "openExternally",
//   "pattern": "^\\s*Now listening on:\\s+(https?://\\S+)"
// },
```

Configure the web application with the Azure AD application

Locate and open the **./appsettings.json** file in the ASP.NET Core project.

Set the `AzureAd.Domain` property to the domain of your Azure AD tenant where you created the Azure AD application (*for example: contoso.onmicrosoft.com*).

Set the `AzureAd.TenantId` property to the **Directory (tenant) ID** you copied when creating the Azure AD application in the previous section.

Set the `AzureAd.ClientId` property to the **Application (client) ID** you copied when creating the Azure AD application in the previous section.

Request Authentication & Bearer Token validation

While the scaffolded middleware will validate the token, authorizing a request for a specific controller action is the responsibility of the developer. In this exercise, the presence of scopes in the token is used to authorize the action. If the scope is present, the action is allowed. Validating scopes will be done in the action methods of the controllers that will be added later in this exercise.

Data models and sample data

By convention, .NET Core web API projects store model classes in a folder named Models. Create a new folder named **Models** in the project directory.

In the **Models** folder, create a new file named **Category.cs** and add the follow C# code to it:

C#

```
namespace ProductCatalog.Models
{
  public class Category
  {
    public int Id { get; set; }
    public string Name { get; set; }
  }
}
```

In the **Models** folder, create a new file named **Product.cs** and add the following C# code to it:

C#

```
namespace ProductCatalog.Models
{
  public class Product
  {
    public int Id { get; set; }
    public string Name { get; set; }
    public Category Category { get; set; }
  }
}
```

This exercise will store sample data in-memory while the app is running. The data is randomly generated when the app is started using a NuGet package.

Install the NuGet package by running the following from your command prompt in the project folder:

Console

```
dotnet add package Bogus
```

Return to **Visual Studio Code** and create a new file named **SampleData.cs** in the root folder of the project. Add the following C# code to the file:

C#

```
using System.Collections.Generic;
using Bogus;
using ProductCatalog.Models;
```

```
namespace ProductCatalog
{
  public class SampleData
  {
    public List<Category> Categories { get; set; }
    public List<Product> Products { get; set; }

    public static SampleData Initialize()
    {
      var data = new SampleData();

      var categoryIds = 0;
      var categoryFaker = new Faker<Category>()
        .StrictMode(true)
        .RuleFor(c => c.Id, f => ++categoryIds)
        .RuleFor(c => c.Name, f => f.Commerce.Categories(1)[0]);
      data.Categories = categoryFaker.Generate(10);

      var productIds = 0;
      var productFaker = new Faker<Product>()
        .StrictMode(true)
        .RuleFor(p => p.Id, f => ++productIds)
        .RuleFor(p => p.Name, f => f.Commerce.Product())
        .RuleFor(p => p.Category, f => f.PickRandom(data.Categories));
      data.Products = productFaker.Generate(20);

      return data;
    }

  }
}
```

The sample data will be stored as a singleton in the dependency injection container built into ASP.NET Core. Open the **Startup.cs** file in the root folder of the project. Add the following line at the bottom of the `ConfigureServices()` method:

C#

```
services.AddSingleton(SampleData.Initialize());
```

Web API Controllers

By convention, .NET Core WebAPI projects store controller classes in a folder named Controllers. In the **Controllers** folder, create a new file named **CategoriesController.cs** and add the follow C# code. The controller has the `[Authorize]` attribute, which forces the request to have a valid

Authorization header in the request. Each action method makes a call to the `VerifyUserHasAnyAcceptedScope()` method, specifying the scope required to execute the action:

C#

```
using System.Collections.Generic;
using System.Linq;
using ProductCatalog.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Identity.Web.Resource;

namespace ProductCatalog.Controllers
{
  [Authorize]
  [ApiController]
  [Route("api/[controller]")]
  public class CategoriesController : ControllerBase
  {
    SampleData data;

    public CategoriesController(SampleData data)
    {
      this.data = data;
    }

    public List<Category> GetAllCategories()
    {

      HttpContext.VerifyUserHasAnyAcceptedScope(new string[] { "Category.Read" });
      return data.Categories;
    }

    [HttpGet("{id}")]
    public Category GetCategory(int id)
    {
      HttpContext.VerifyUserHasAnyAcceptedScope(new string[] { "Category.Read" });
      return data.Categories.FirstOrDefault(p => p.Id.Equals(id));
    }

    [HttpPost]
    public ActionResult CreateCategory([FromBody] Category newCategory)
    {
      HttpContext.VerifyUserHasAnyAcceptedScope(new string[] { "Category.Write" });
      if (string.IsNullOrEmpty(newCategory.Name))
      {
        return BadRequest("Product Name cannot be empty");
      }
      newCategory.Id = (data.Categories.Max(c => c.Id) + 1);
      data.Categories.Add(newCategory);
```

```
      return CreatedAtAction(nameof(GetCategory), new { id = newCategory.Id }, new-
Category);
    }
  }
}
```

In the **Controllers** folder, create a new file named **ProductsController.cs** and add the follow C# code:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ProductCatalog.Models;
using Microsoft.Identity.Web.Resource;

namespace ProductCatalog.Controllers
{
  [Authorize]
  [ApiController]
  [Route("api/[controller]")]
  public class ProductsController : ControllerBase
  {
    SampleData data;


    public ProductsController(SampleData data)
    {
      this.data = data;
    }

    public List<Product> GetAllProducts()
    {
      HttpContext.VerifyUserHasAnyAcceptedScope(new string[] { "Product.Read" });
      return data.Products;
    }

    [HttpGet("{id}")]
    public Product GetProduct(int id)
    {
      HttpContext.VerifyUserHasAnyAcceptedScope(new string[] { "Product.Read" });
      return data.Products.FirstOrDefault(p => p.Id.Equals(id));
    }

    [HttpPost]
    public ActionResult CreateProduct([FromBody] Product newProduct)
```

```
    {
      HttpContext.VerifyUserHasAnyAcceptedScope(new string[] { "Product.Write" });
      if (string.IsNullOrEmpty(newProduct.Name))
      {
        return BadRequest("Product Name cannot be empty");
      }

      newProduct.Category.Name = data.Categories.FirstOrDefault(c => c.Id ==
newProduct.Category.Id)?.Name;
      if (string.IsNullOrEmpty(newProduct.Category?.Name))
      {
        return BadRequest("Product Category cannot be empty");
      }
      newProduct.Id = (data.Products.Max(p => p.Id) + 1);
      data.Products.Add(newProduct);
      return CreatedAtAction(nameof(GetProduct), new { id = newProduct.Id }, newProd-
uct);
    }
  }
}
```

You now have a web API that is secured with Microsoft identity. Additional exercises in this module demonstrate different types of applications that can authenticate and call this web API.

Summary

In this exercise, you learned how to create a .NET web API application and secure it with Microsoft identity.

Test your knowledge

1. Developers can use Microsoft identity to secure which types of apps?

- ○ Web applications
- ○ Desktop and mobile applications
- ○ Web APIs
- ○ All of these applications can be secured with Microsoft identity

2. What types of authentication flows can developers use when securing a web API with Microsoft identity?

- ○ Only the on behalf of flow is supported by Microsoft identity with web APIs.
- ○ Only applications that access the web API as themselves, not as a user.
- ○ Both applications that access the web API without a user and the on behalf of flow are supported.

Check your answers

How are we doing? ☆ ☆ ☆ ☆ ☆