**Lab: Web API**

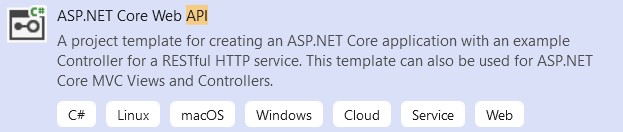
Lab for the ["ASP.NET Advanced" course @ SoftUni](https://softuni.bg/trainings/4107/asp-net-advanced-june-2023)

In this task, we will create a **simple REST API** for **displaying**, **creating**, **editing** and **deleting products**. We will try out the **API functionalities** with the help of the **Postman** **tool**. At the end, we will use the **Swagger** **tool** to **document** **our API** and **try it out** directly from the browser.

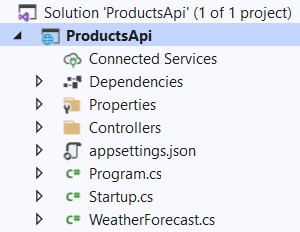
**Web API for Products**

# Step 1: Create and Clean the Project

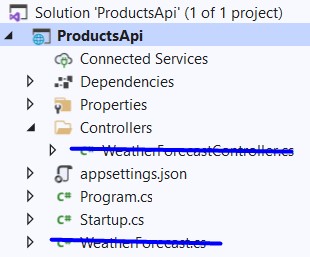
Let's start with **creating the API**. Open **Visual** **Studio** and choose the "**ASP.NET** **Core** **Web** **API**" **template**:



**Create the API** with the current **.NET** **version**. It does **not need authentication**! When created, the **solution** should look like this:

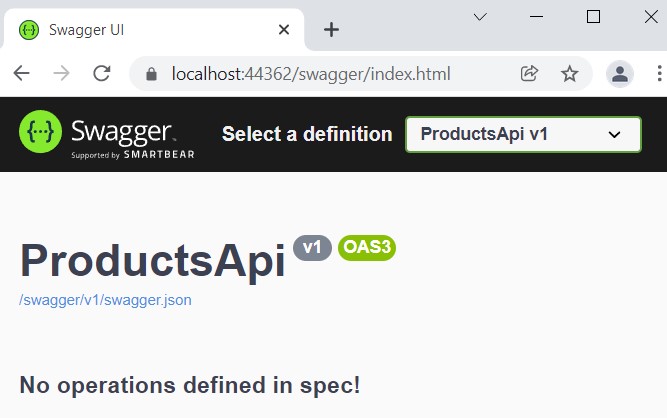


As you can see, we have an **API controller** for **weather forecast**. We won't need it, so **delete** the **WeatherForecastController** and **WeatherForecast** **classes**:



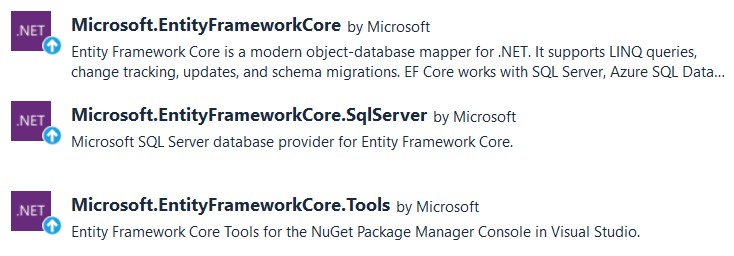
Now you can **run the app**. You will see that we have **Swagger** as a **part of our API** (coming from the **template**), but it does not show anything, as we **do not have any controllers** and actions yet:

**Step 2: Create a Database**



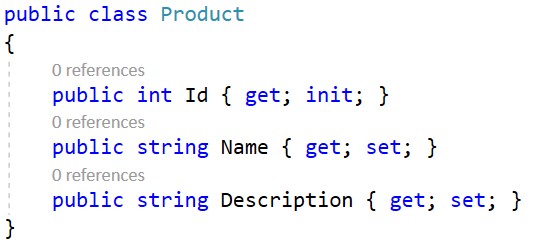
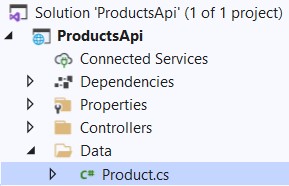
We will need to **create a database** for our **products**. However, you can see that we **do not have a db context** and we should **create it** from scratch.

To begin with, **download** the **Entity** **Framework** **Core** **NuGet** **packages** we will need for creating the database:



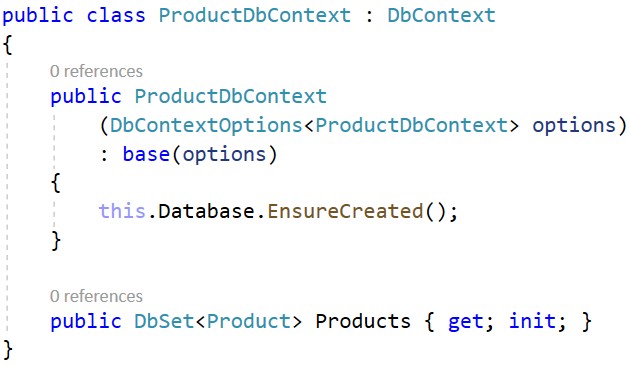
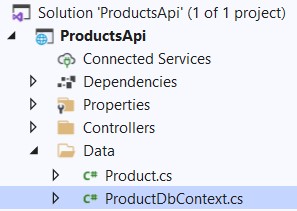
Then, create a **folder** "**Data**", which will hold the data-related classes. In it, **create the** **Product** **class**, which should have **properties for id**, **name** and **description**:

→



Then, create the **ProductDbContext** **class** in the "**Data**" **folder**. It should have a **DbSet** for **products** and should **invoke the** **EnsureCreated()** **method in the constructor**. In this way, we will **create the database** but we won't be able to change it, unless we **apply new migrations** to the database **manually**:

→



Next, we should **add a connection string** to the **appsettings.json** **file**, so that we can **connect to SQL Server**. Add the following lines like this:

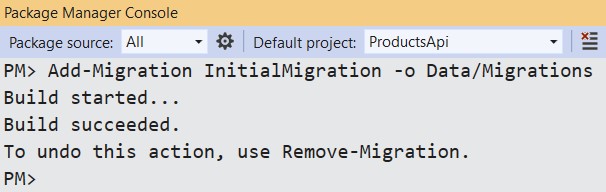


You can **copy the connection string** from here:

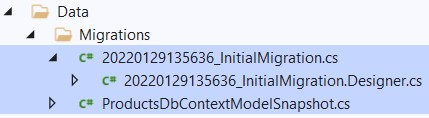
Server=(localdb)\\mssqllocaldb;Database=ProductsDb;Trusted\_Connection=True;MultipleActiveResultSets=true

Next, we should **register the db context class** as a **service**. Do this in the **Program** **class**, you should already know how to do that.

At the end, we should just **add a migration**, which will be applied, so that our **database is created**. To do this, **open** the **Package** **Manager** **Console** from **[Tools]** → **[NuGet** **Package** **Manager]** and **add the migration** like this:



The **migration** should appear in the "**/Data/Migrations**" **folder**:



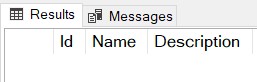
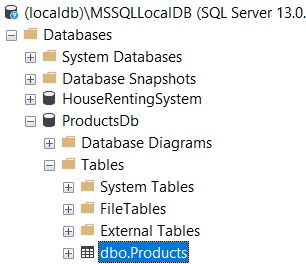
**Run your app** again. Open **SQL** **Server** **Management** **Studio** and you should see the **newly-created database**. It has a single **table** "**Products**":

→

Now you can start building your

**API controller**

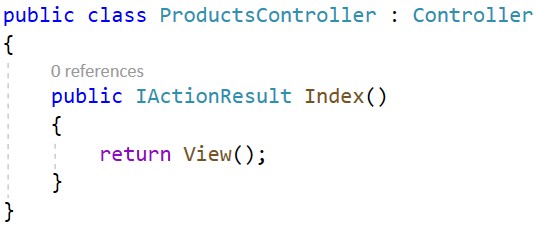
.



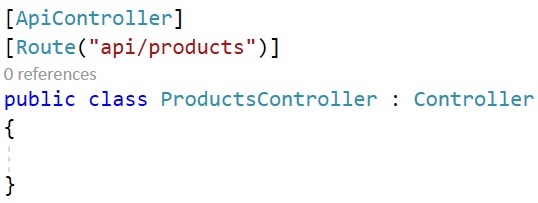
# Step 3: Create the API Controller Class

To **create an API controller**, you should first **create a standard controller class** and **add the needed attributes**. Create the **ProductsController** **class** in the "**Controllers**" **folder**:

→



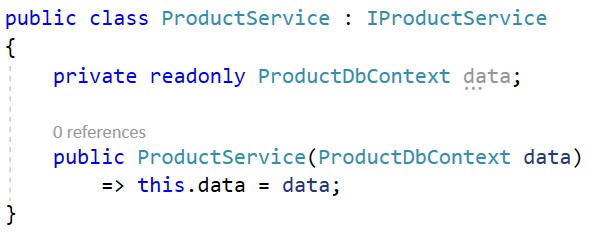
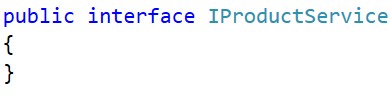
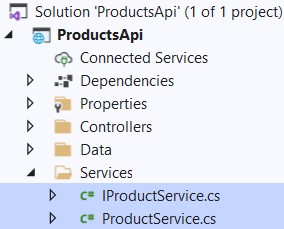
**Remove** **the Index()** **method**, as we won't need it. Then, add the **[ApiController]** and **[Route]** **attributes** to make the controller and **API controller**:



Note that our controller methods will be **accessed** on "**/api/products**" because this is how we set it in the **[Route]** **attribute**.

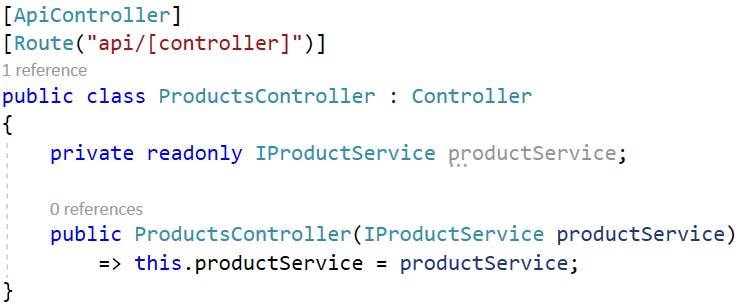
We want our **business logic** to be implemented in **service methods** and the **controller to use them directly**. For this reason, we should create a new **folder** "**Services**" with an **IProductService** **interface** and an **ProductService** **class**. The class should accept the **db** **context** from the **constructor**:

→



Don't forget to **register the service** in the **Program class**.

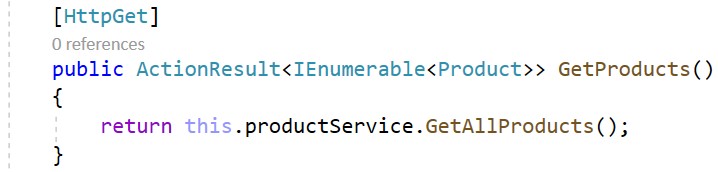
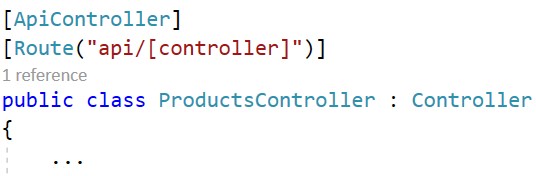
Now go back to the **ProductsController** **class** and **inject the created service**:



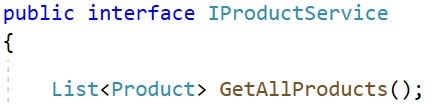
# Step 4: Write the API Controller Methods

## GetProducts() Method

The first method should **return all products** as an **ActionResult** with a **collection of type Product** (we won't create and return a model, as we have a pretty basic class for the product). The **controller method** should use a **service method** and should be i**nvoked on a "GET" request to "/api/products"**. Do it like this:



The **IProductService** and **ProductService** **classes** define and implement the **GetAllProducts()** **method**:





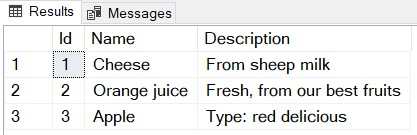
**Run** the app. Then, open **Postman** and **create a "GET" request to "/api/products"** (use the **whole URL**):



The **returned response** should be with **status code** "**200** **OK**" but will return an **empty JSON object**, as we don't have **any products in our database** yet:

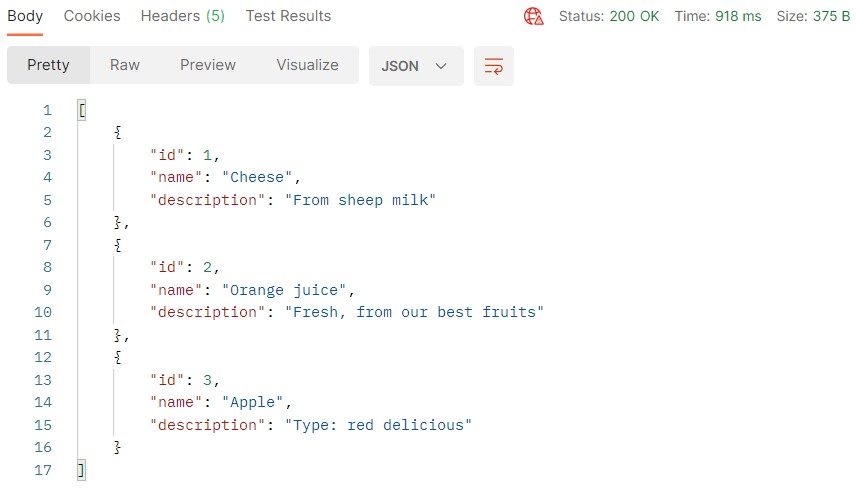


You can **open** **SQL** **Server** **Management** **Studio** and **add some products** to display:

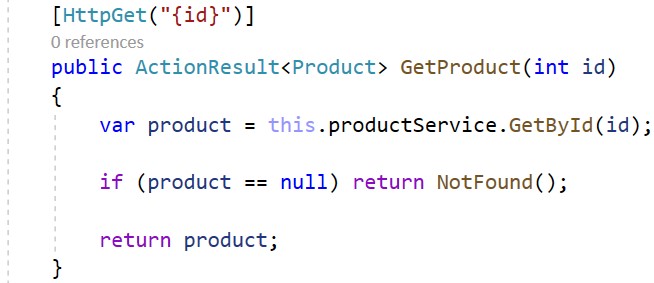
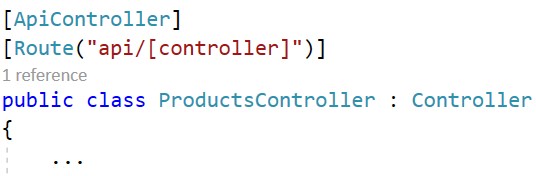


Then, if you **send the request** in **Postman** again, you should see the above **products returned as JSON**:

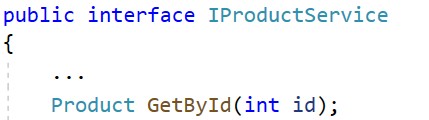
**GetProduct() Method**



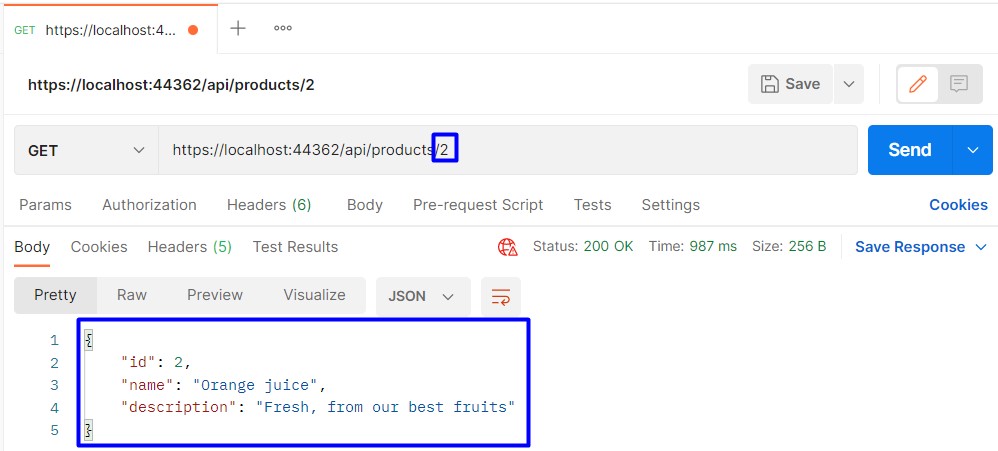
The **GetProduct(int** **id)** method should **return a product by a given id** if it exists. If it **doesn't exist**, a "**404** **Not** **Found**" **response** should be returned. The method should be **invoked on a "GET" request to "/api/products/{id}"**. Write it in the **ProductsController** **class** like this:



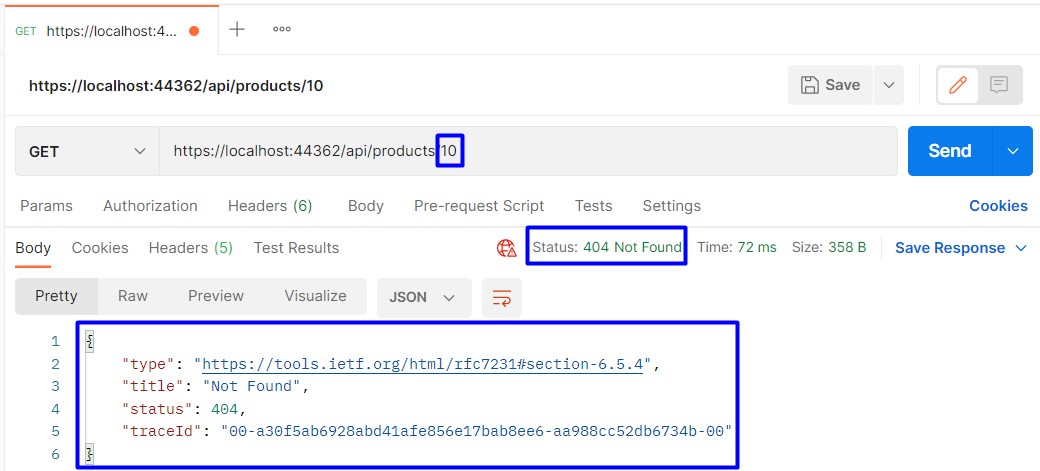
The **GetById()** **service method** should look like this:



Now **try out the method** in **Postman** by sending a "**GET**" **request** to "**/api/products/{id}**". If you **send an id** of an **existing product**, the **product should be returned**:

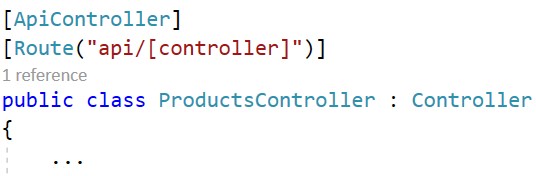


If you **send an invalid id**, a "**404** **Not** **Found**" **response** should be **returned** (again as **JSON**):

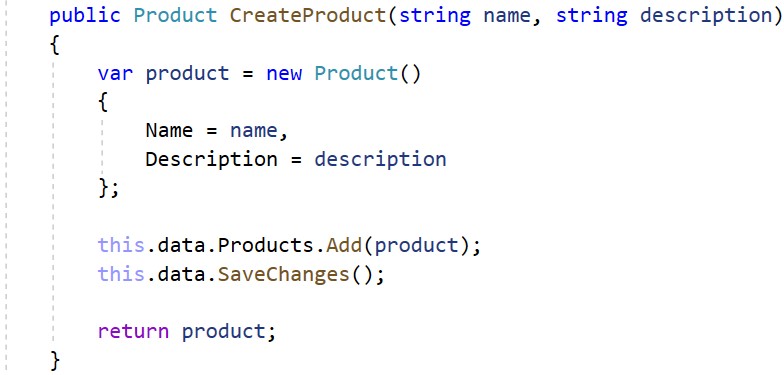
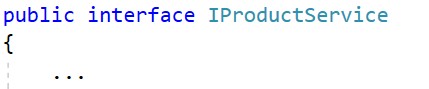


## PostProduct() Method

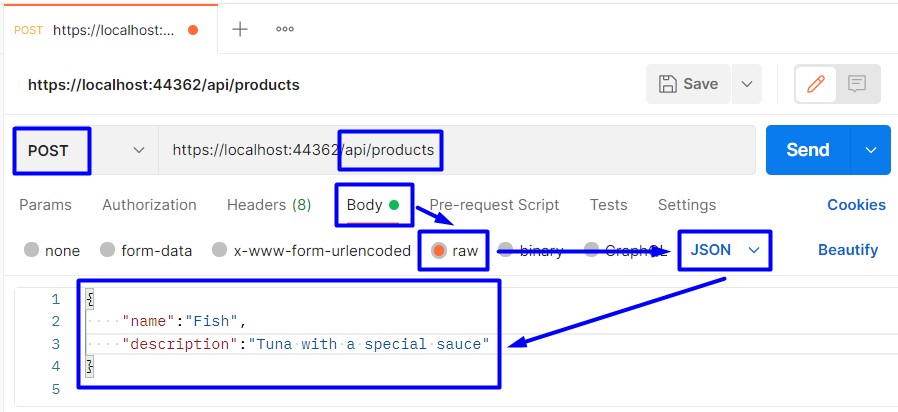
The **PostProduct(Product** **product)** **controller method** is responsible for **creating a new product** in the database. When the **product is created**, a "**201** **Created**" **response** should be returned, which will invoke the **GetProduct(int** **id)** **method** to **return the product**. It should be invoked on a "**POST**" **request** to "**/api/products**":



The **CreateProduct()** **service method** is the following:



**Run the app** and **try to create a new product** in **Postman**. To do this, you should send a "**POST**" **request** to "**/api/products**" and **add a body to the request** with the **new product**. The **body** should be in a **raw JSON format**. Do it like this:



The **response** should be the following if the **product is created successfully**:

The

**new**

**product**

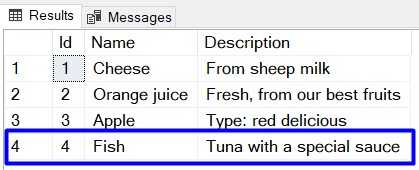
should appear in the

**database**

:

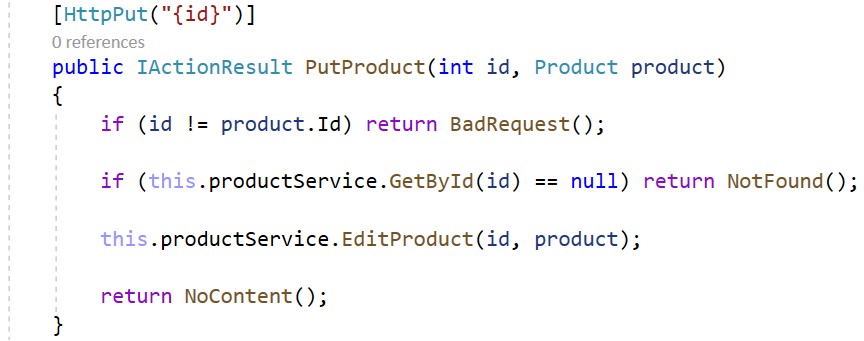
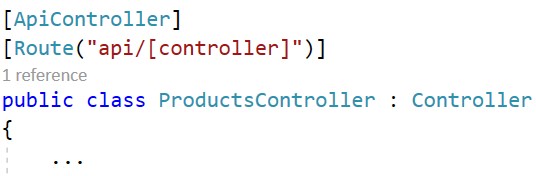


**PutProduct() Method**

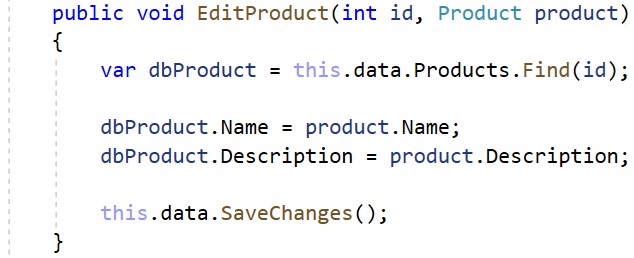
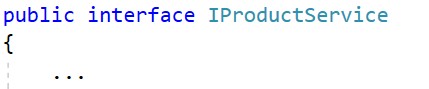


The **PutProduct(int** **id,** **Product** **product)** **method** of the **ProductsController** should be invoked on a "**PUT**" **request** to "**/api/products/{id}**" with the **data of the product** (**modified** and **non-modified**). If the product id from the **URL** and from the **request body are not the same**, a "**400** **Bad** **Request**" **response** is returned. If a **product with the** **given id does not exist**, a "**404** **Not** **Found**" **response** is returned. If the product is **edited** **successfully**, a "**204** **No** **Content**" **response** is returned.

Write the action like this:

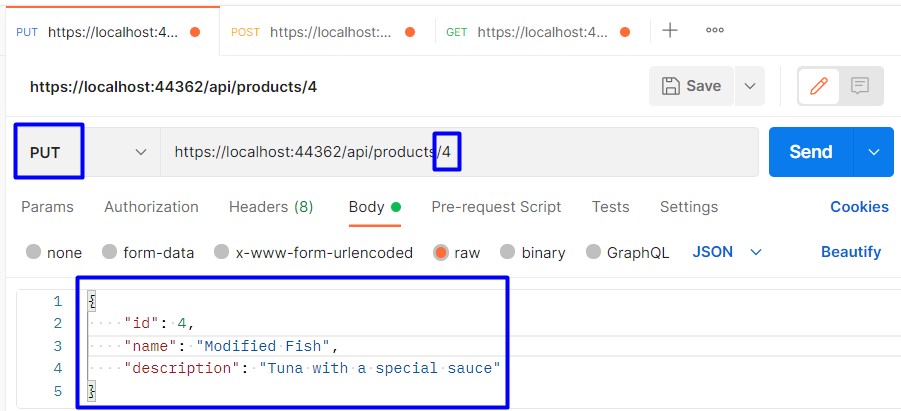


The **EditProduct(…)** **service method** is shown below:



Try to **edit the product** we created in **Postman**. Send a "**PUT**" **request** to "**/api/products/{id}**" with an **existing** **product id** and with the **product data**. Note that you should include **all the product data** in the **request body**, no matter if it is modified or not. If you **miss a property**, a **NULL** **value** will be assigned to it.

Now **edit an existing product** in **Postman** like this:



If the **edit is successful**, an **empty** "**204** **No** **Content**" **response** should be returned:

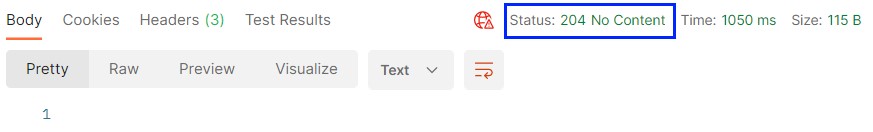
The

**product**

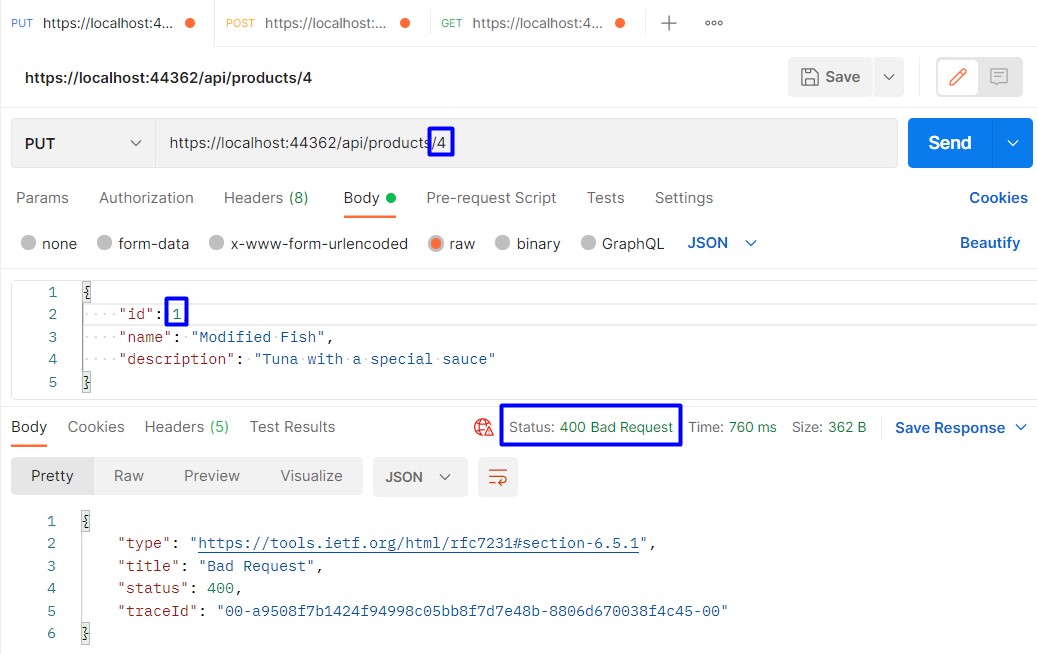
should be

**modified in the database**

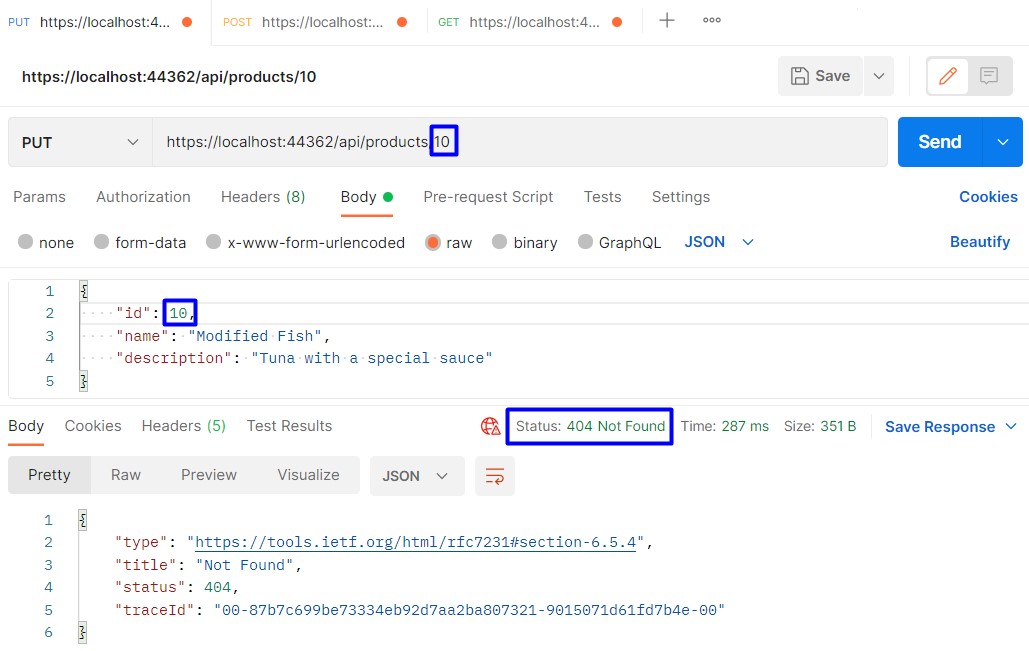
, as well:



If you **send a request** in **Postman** with **different ids in the URL and in the body**, a "**400** **Bad** **Request**" should be returned:



If you **send a product with an id**, which **does not exist** in the database, a "**404** **Not** **Found**" **response** should be returned:

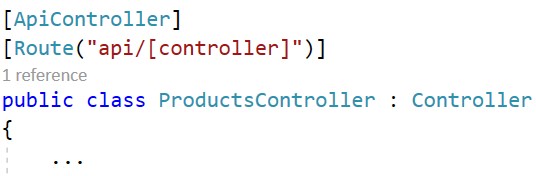


## PatchProduct() Method

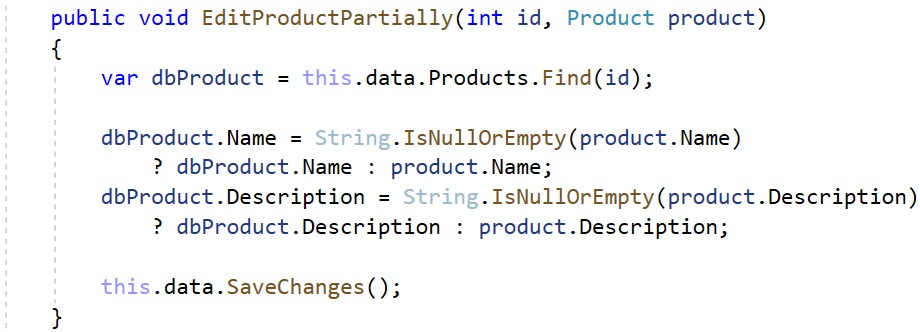
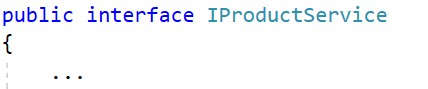
The **PatchProduct()** **method** is pretty **similar** to the **PutProduct()** one we created. The difference is that when you send a "**PUT**" **request**, the **request body should contain the whole product data**, while the "**PATCH**" **request** body should **only have the modified property values**.

The **PatchProduct(int** **id,** **Product** **product)** **method** of the **ProductsController** should be invoked on a "**PATCH**" **request** to "**/api/products/{id}**" with **partial** **data of the product** (**only** **modified**). If a **product with the** **given id does not exist**, a "**404** **Not** **Found**" **response** is returned. If the product is **edited** **successfully**, a "**204** **No** **Content**" **response** is returned.

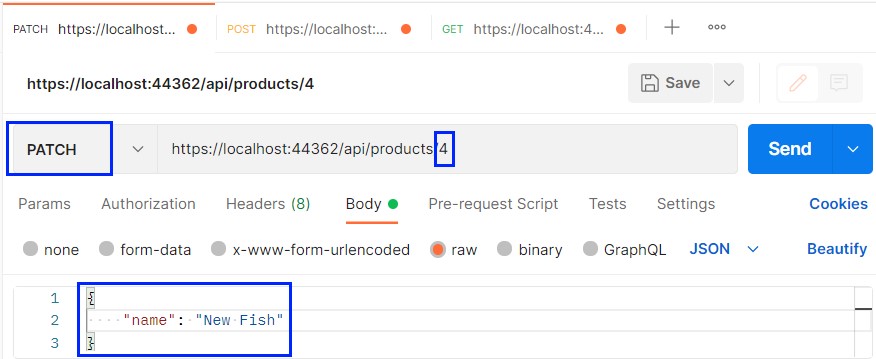
Write the action like this:



The **EditProductPartially()** **service method** should **check model properties for null values** and **modify some fields with the provided data**:



Now create a "**PATCH**" **request** in **Postman** to "**/api/products/{id}**" with a **valid id** and with **modified data** **only**:

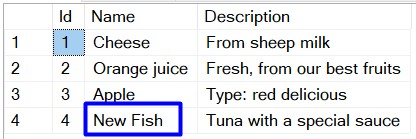
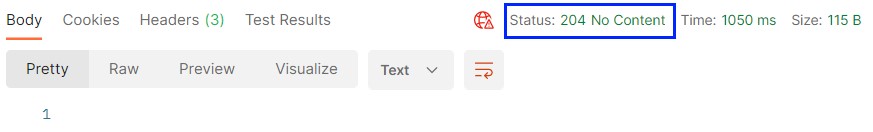


The **product title** should be **modified** successfully and a "**204** **No** **Content**" **response** should be returned:

Check the

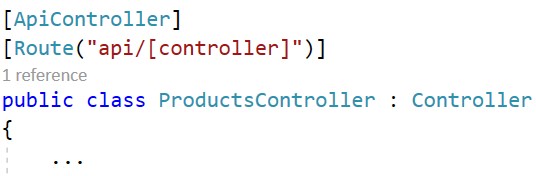
**modification in the database**

, as well:

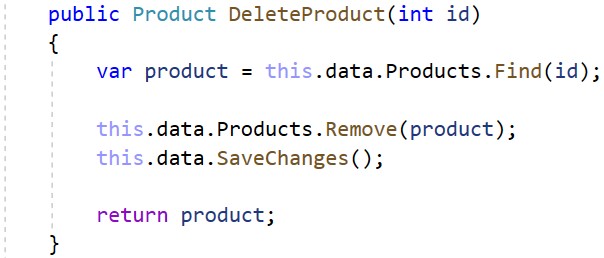
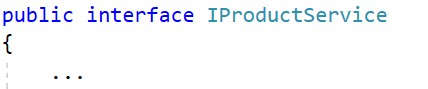


## DeleteProduct() Method

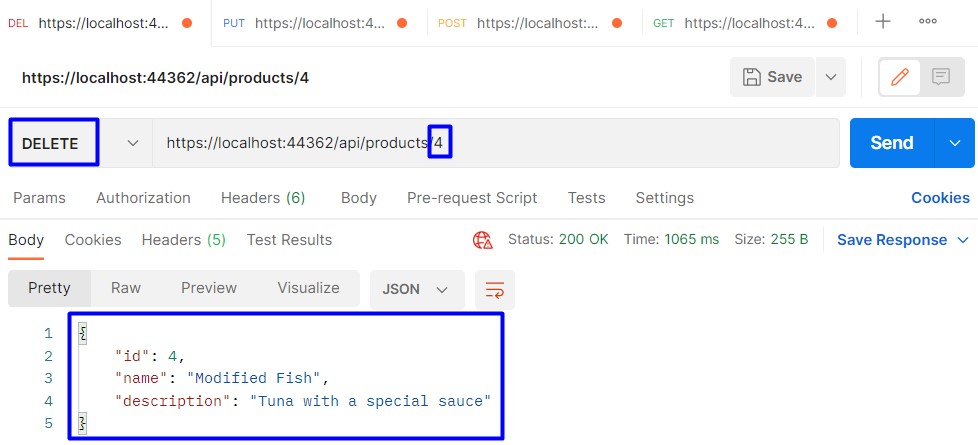
The **DeleteProduct(int** **id)** **method** is the last method we will implement. It should be invoked on a "**DELETE**" **request** to "**/api/products/{id}**". If a **product with the given id doesn't exist**, "**404** **Not** **Found**" is returned. If it **exists**, the **deleted product is returned**:



The **DeleteProduct(…)** **service method** is the following:



Try to **delete the product** we created in **Postman**. Create the following **request** and make sure that the **product is** **returned in the response**:

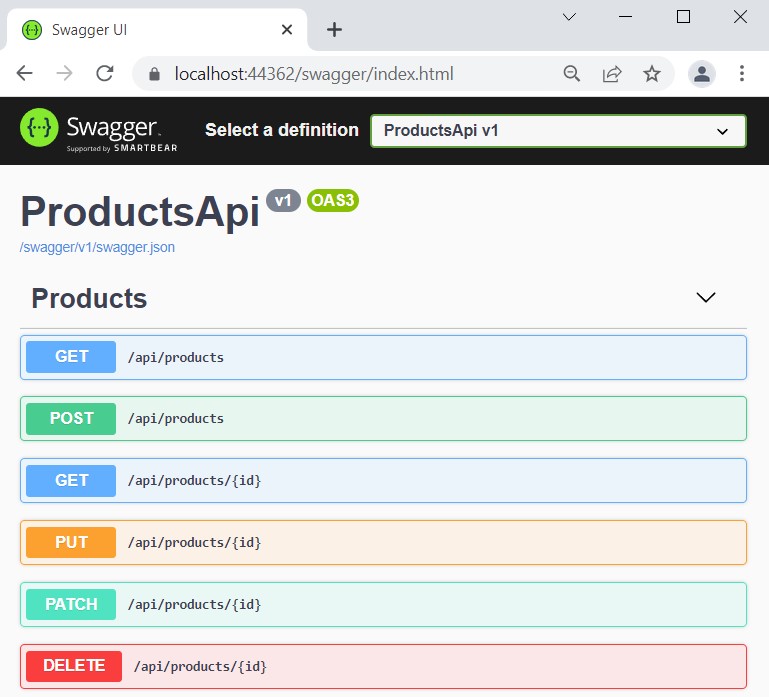


Now you have an **implemented REST API** with **ASP.NET Core**.

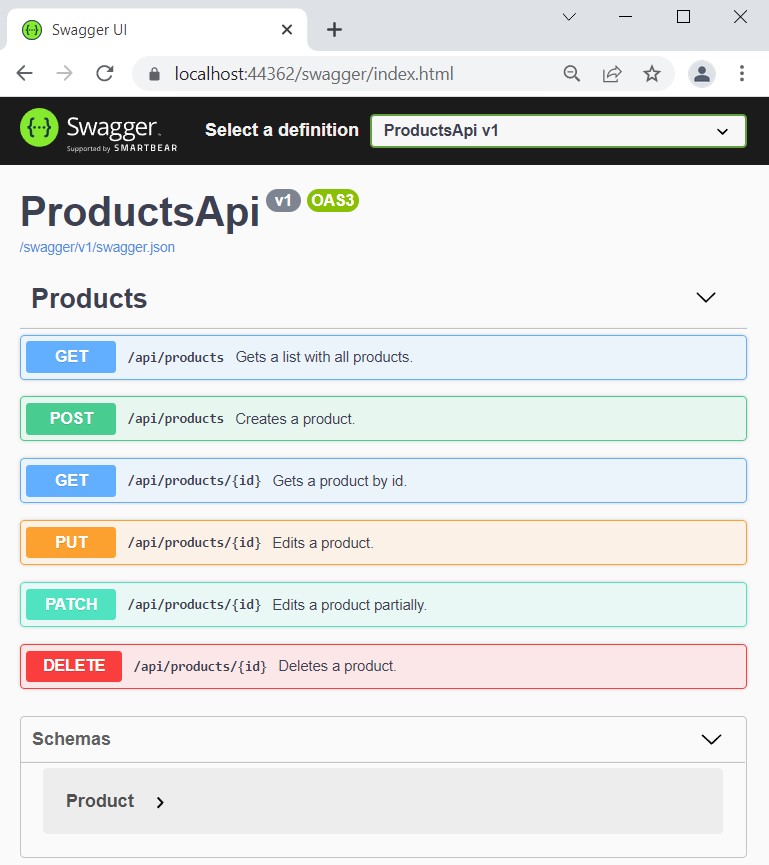
# Step 4: Write the API Swagger Documentation and Try It Out

Finally, we will see how to write **OpenAPI** **documentation** for **Swagger**, so that it **displays correct and full** **information** about our **API methods**.

To begin with, if you **run the app** we created, you will see that **Swagger already** **displays** **our** **API controller methods**:

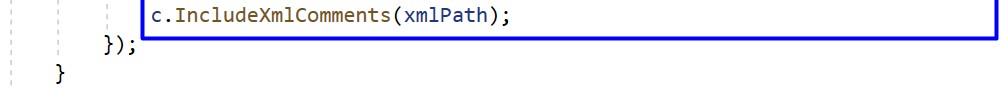


After we **add documentation**, the **Swagger** **page** will look like this:

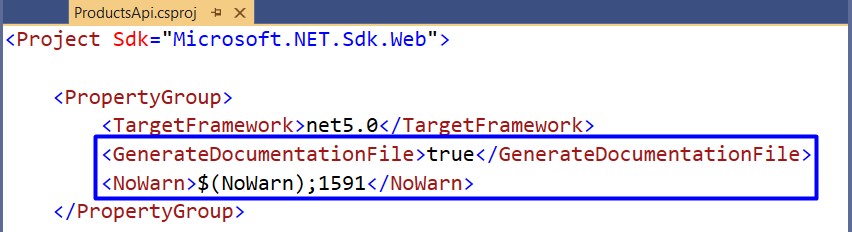


To do this, we should first **enable XML comments** (the ones you see on each method). These comments will be **saved in an XML file** as a part of your project. To **create such a file**, go to the **ConfigureServices(…)** **method** of your **Startup** **class** and **add the following lines** to the **default** **Swagger** **options**:

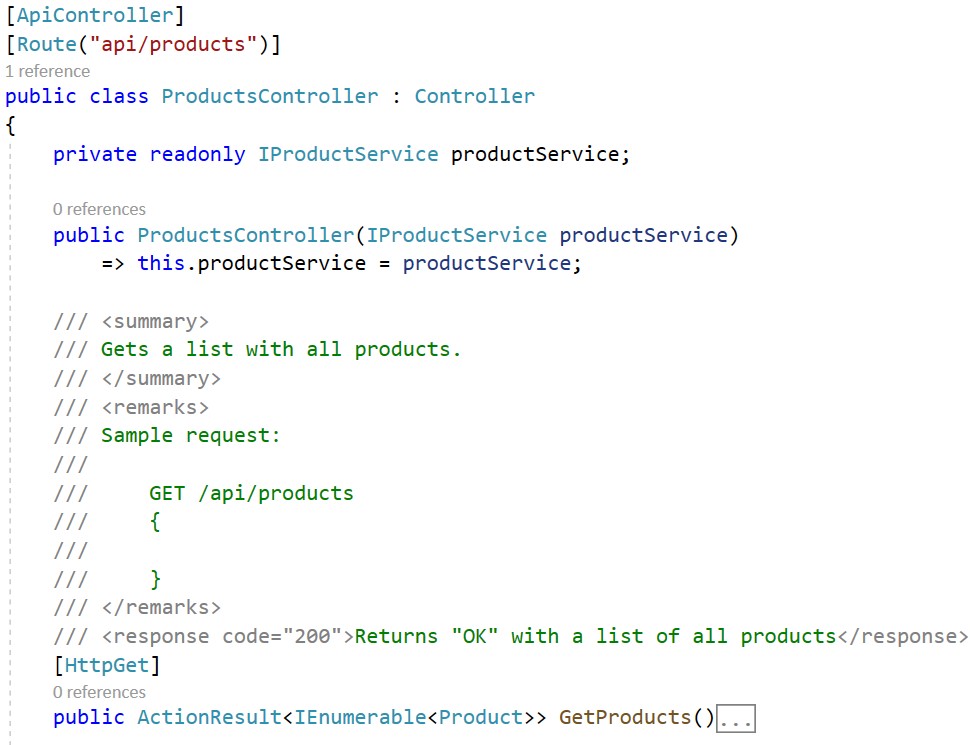


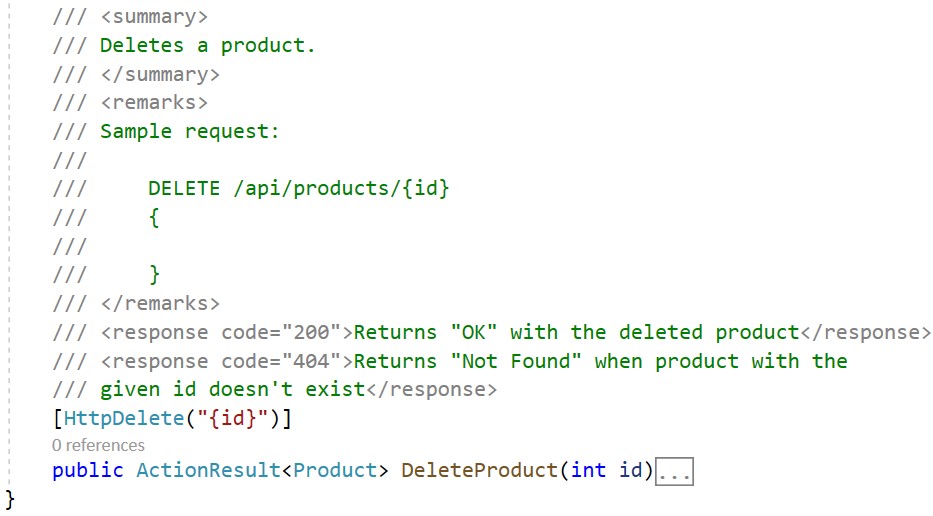
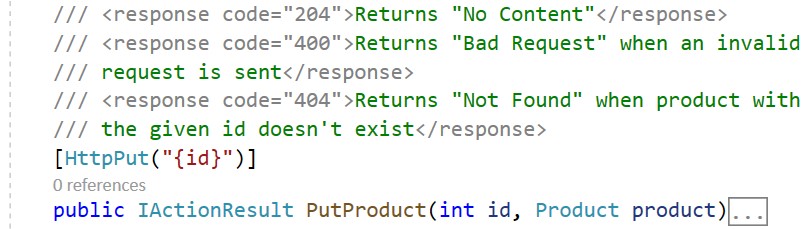
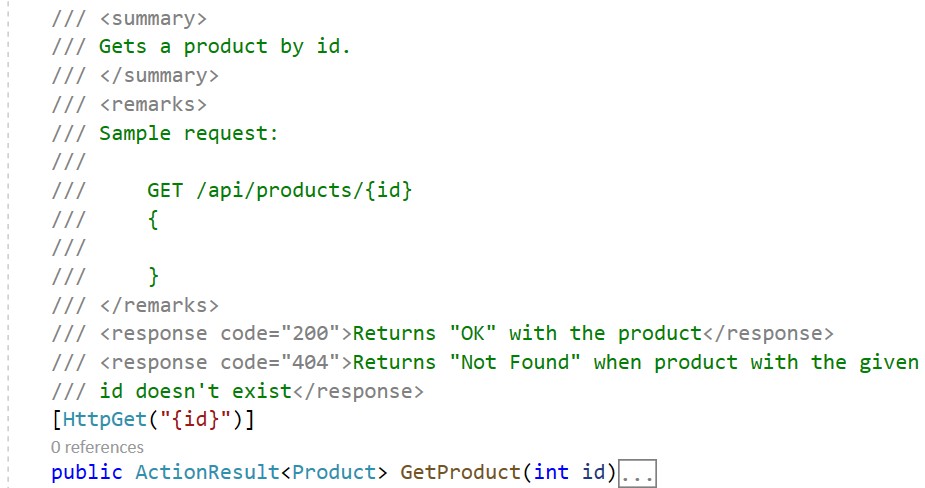


Next, we should **go to the** **.csproj** **file** of our project and **suppress warning messages**, which indicate undocumented types and members. Also, we want our **XML file to be created**, so **add the following lines**:



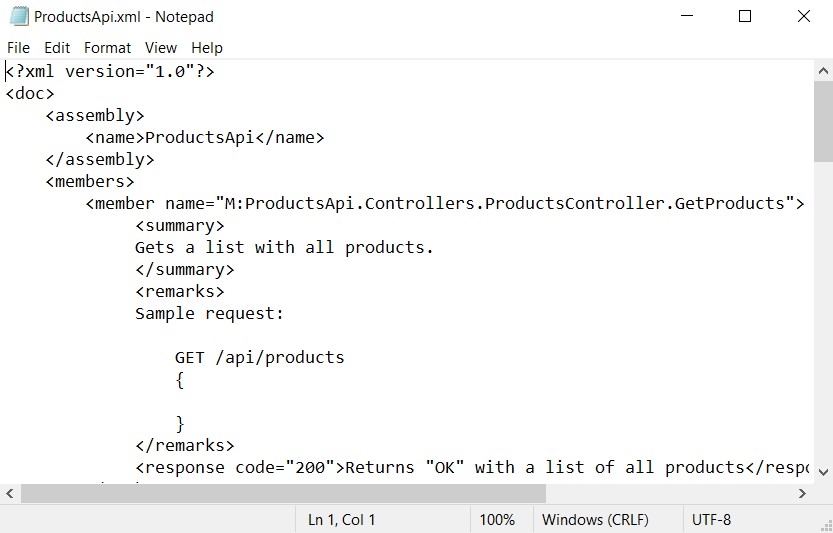
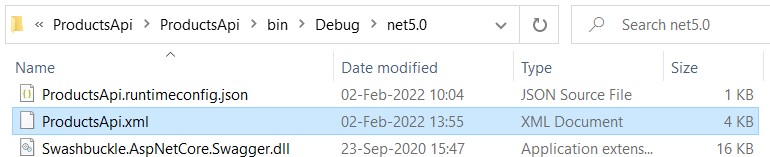
As we **have the file**, let's **add the documentation**, which it will contain. To do this, we will **add triple-slash comments** to **ProductsController** **actions** with a **summary** of what the action does, a **sample request** and the **responses**. Do it like this:



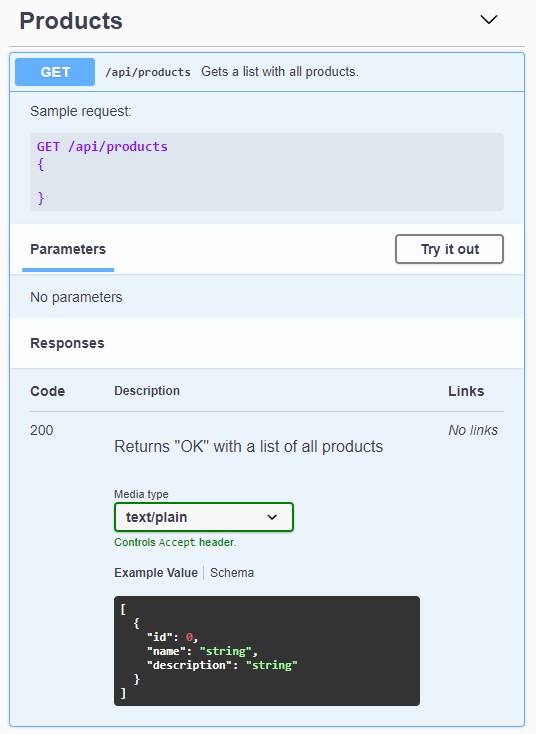


Now **run the app**. Go to your **project's folder** in **File** **Explorer** and navigate to "**bin**" → "**Debug**" → "**net5.0**" and you should see the **generated XML file** with the **documentation**:

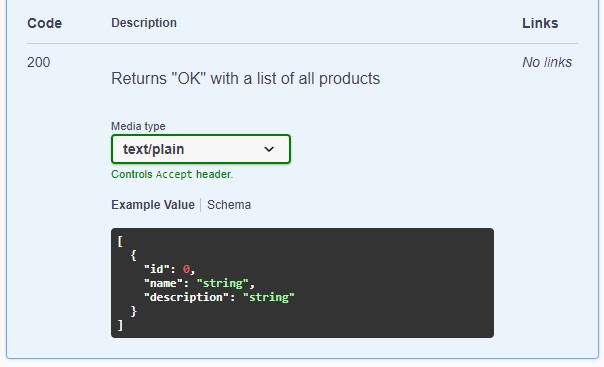
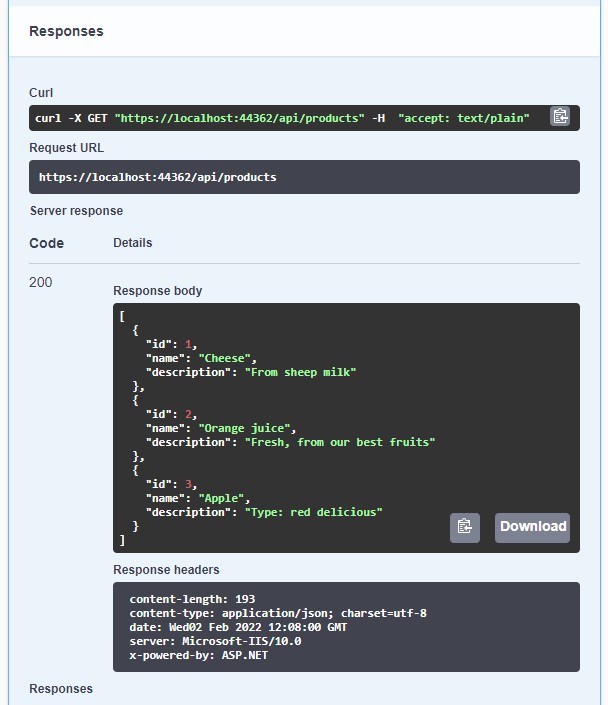
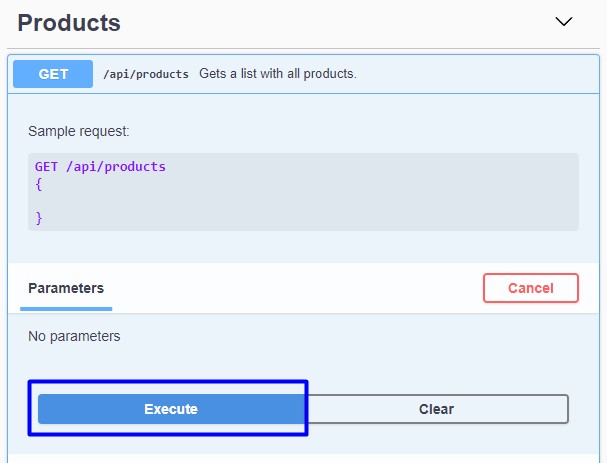
→



The **app in the browser** should have **these comment on the methods**. For example, this is the "**GET**" **method** on "**/api/products**":



Also, **Swagger** gives you the opportunity to **try out the methods directly**. To do this, you should **click** on the **[Try** **it out]** **button**, add an **URL parameter** or **request body data** if needed, and **click** on the **[Execute]** **button**. Then, you should see the **response**:



**Try out** the other methods, too. You should be able to **read**, **create**, **edit** and **delete products**.