Lab: Web API

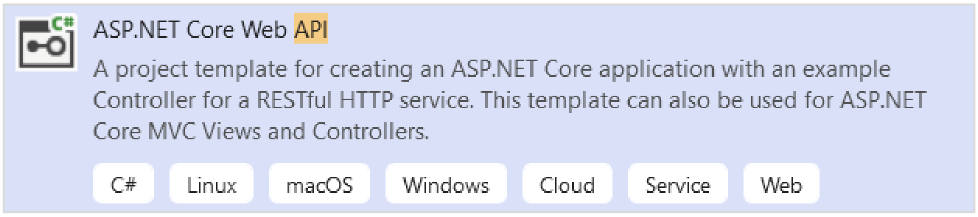
Lab for the "ASP.NET Advanced" course @ SoftUni

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

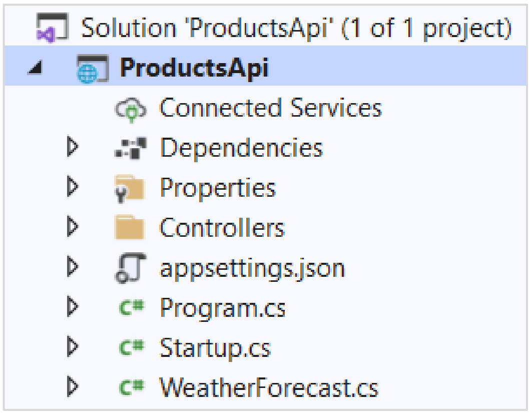
Let's start with **creating the API**. Open **`Visual Studio`** and choose the "**`ASP.NET Core Web API`**" **template**:



ASP.NET Core Web API

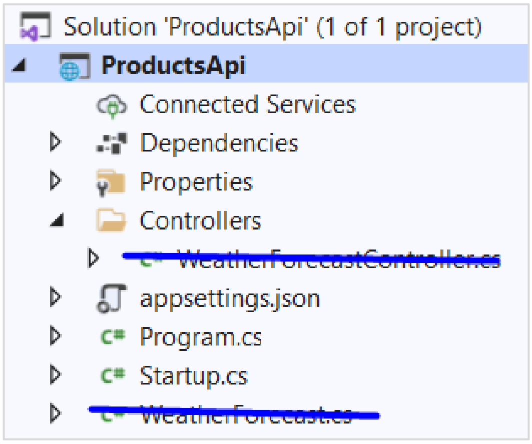
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

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



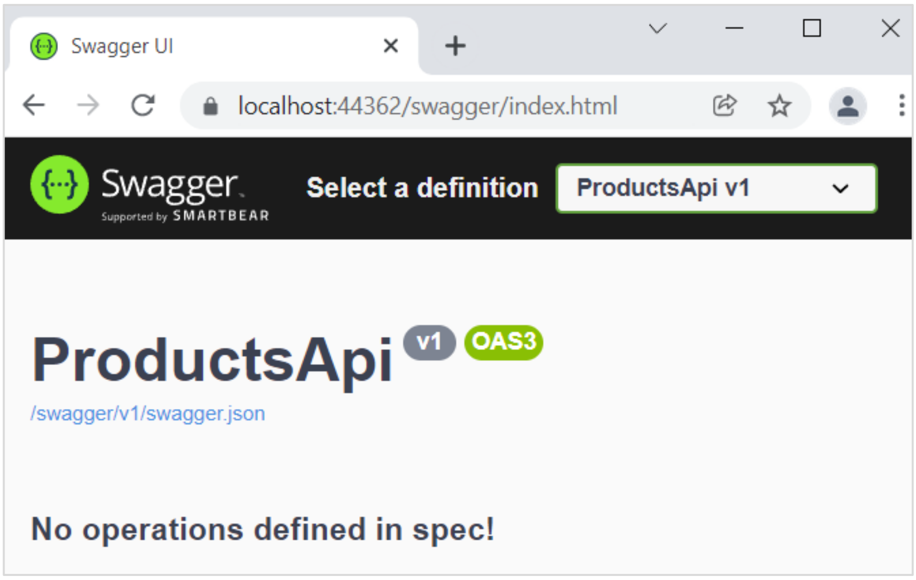
Solution 'ProductsApi' (1 of 1 project)
- ProductsApi
  - Connected Services
  - Dependencies
  - Properties
  - Controllers
  - appsettings.json
  - Program.cs
  - Startup.cs
  - WeatherForecast.cs

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



Solution 'ProductsApi' (1 of 1 project)
- ProductsApi
  - Connected Services
  - Dependencies
  - Properties
  - Controllers
    - ~~WeatherForecastController.cs~~
  - appsettings.json
  - Program.cs
  - Startup.cs
  - ~~WeatherForecast.cs~~

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:

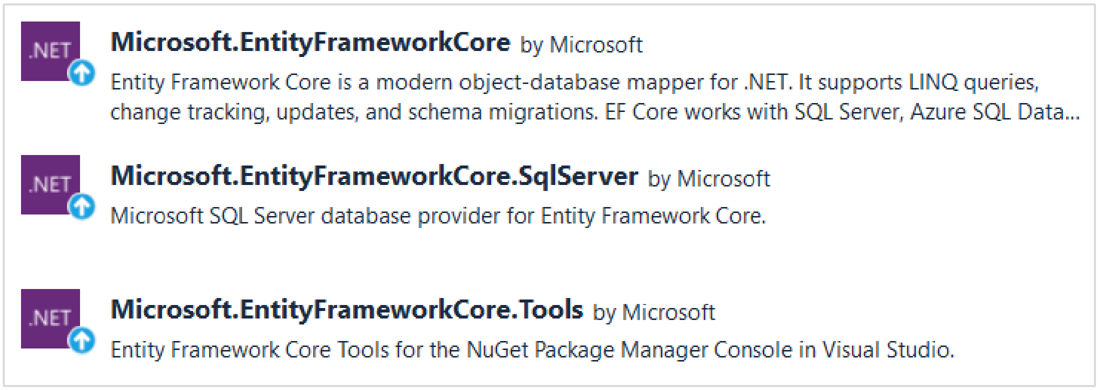


No operations defined in spec!

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:

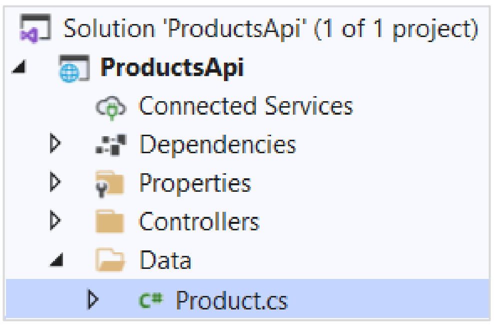


Microsoft.EntityFrameworkCore by Microsoft
Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Data...

Microsoft.EntityFrameworkCore.SqlServer by Microsoft
Microsoft SQL Server database provider for Entity Framework Core.

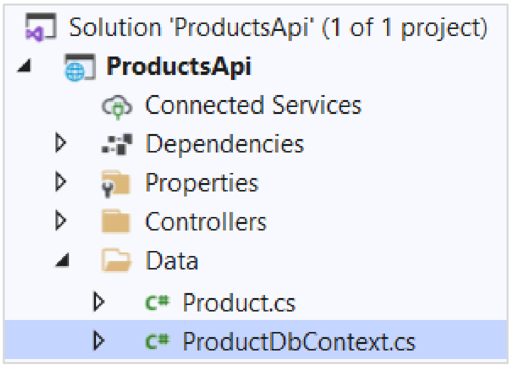
Microsoft.EntityFrameworkCore.Tools by Microsoft
Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.

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**:



```
public class Product
{
    0 references
    public int Id { get; init; }
    0 references
    public string Name { get; set; }
    0 references
    public string Description { get; set; }
}
```

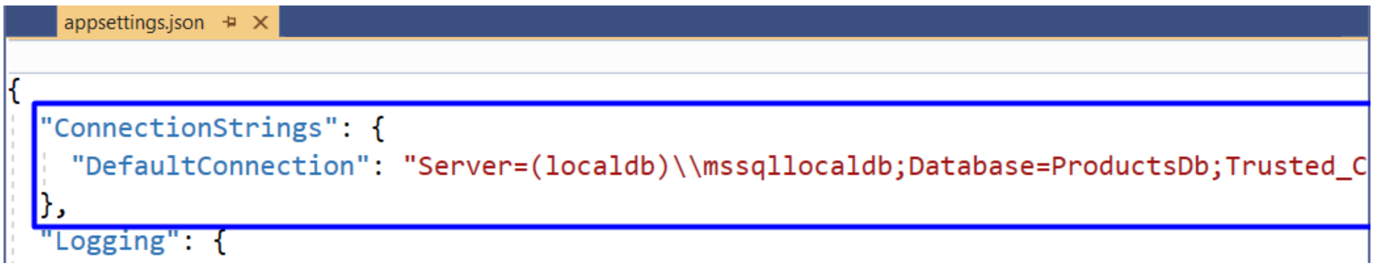
Then, create the **`ProductDbContext` class** in the "**`Data`**" **folder**. It should have a **`DbSet`** for **products** and should **invoke the `EnsureCreated()` method in the constructor**. In this way, we will **create the database** but we won't be able to change it, unless we **apply new migrations**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


Solution 'ProductsApi' (1 of 1 project)
ProductsApi
Connected Services
Dependencies
Properties
Controllers
Data
Product.cs
ProductDbContext.cs

→

```
public class ProductDbContext : DbContext
{
    0 references
    public ProductDbContext
        (DbContextOptions<ProductDbContext> options)
        : base(options)
    {
        this.Database.EnsureCreated();
    }

    0 references
    public DbSet<Product> Products { get; init; }
}
```

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:


appsettings.json

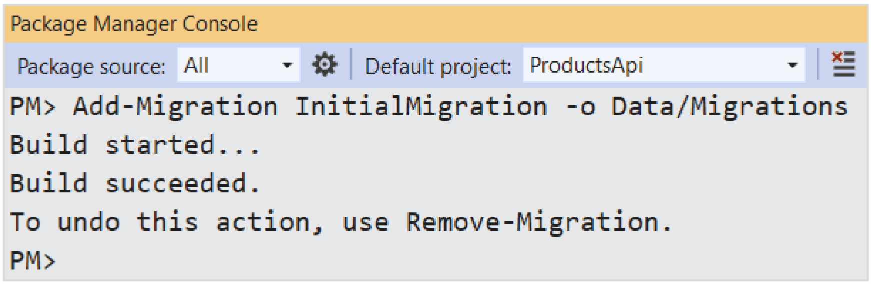
```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ProductsDb;Trusted_C
  },
  "Logging": {
```

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

```
Server=(localdb)\\mssqllocaldb;Database=ProductsDb;Trusted_Connection=True;MultipleActiveResultSets=true
```

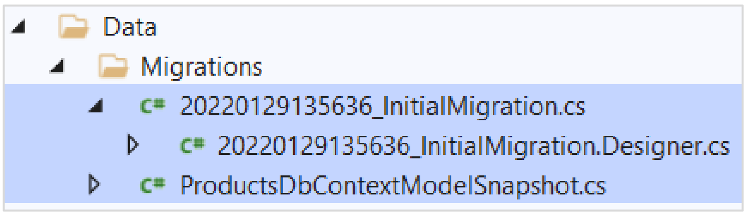
Next, we should **register the db `context` class** as a **service**. Do this in the **`Program` class**, you should already know how to do that.

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:


Package Manager Console
Package source: All
Default project: ProductsApi

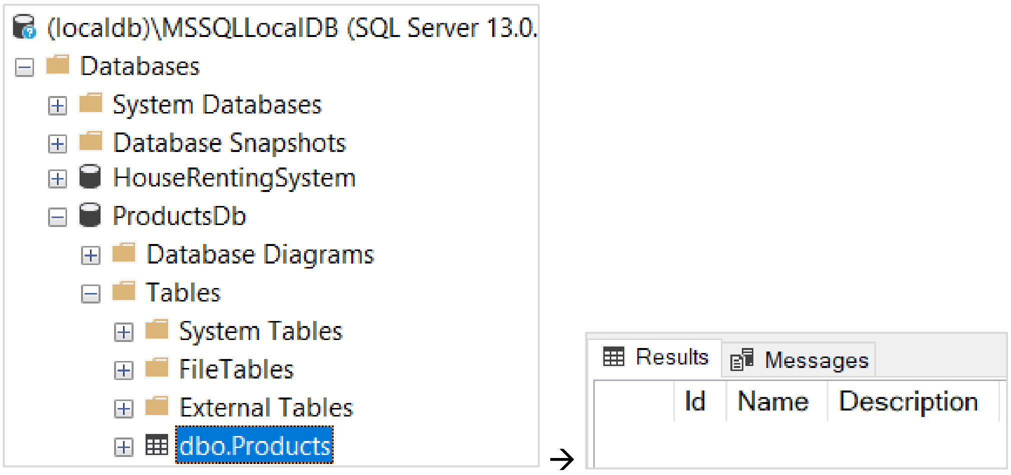
```
PM> Add-Migration InitialMigration -o Data/Migrations
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>
```

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



```
Data
  Migrations
    20220129135636_InitialMigration.cs
      20220129135636_InitialMigration.Designer.cs
    ProductsDbContextModelSnapshot.cs
```

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



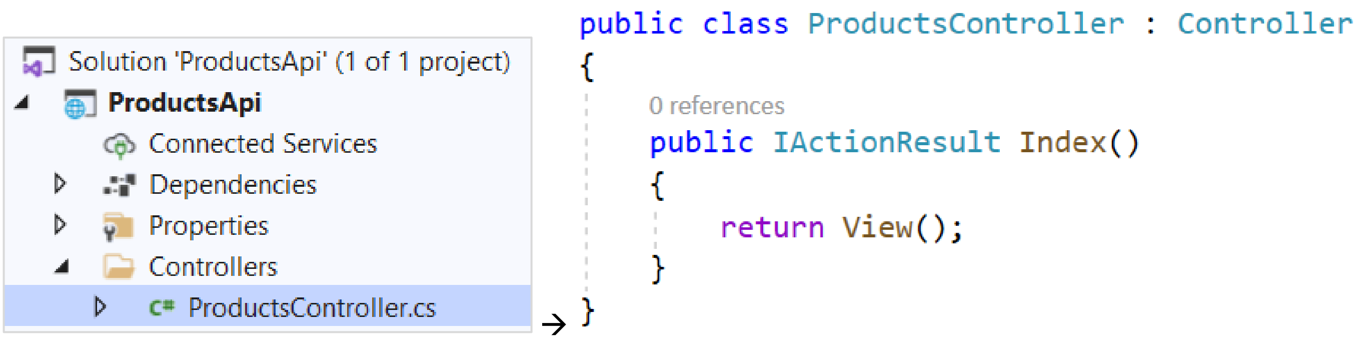
```
(localdb)\MSSQLLocalDB (SQL Server 13.0.
  Databases
    System Databases
    Database Snapshots
    HouseRentingSystem
    ProductsDb
      Database Diagrams
      Tables
        System Tables
        FileTables
        External Tables
        dbo.Products
```

Id	Name	Description

Now you can start building your **API controller**.

Step 3: Create the API Controller Class

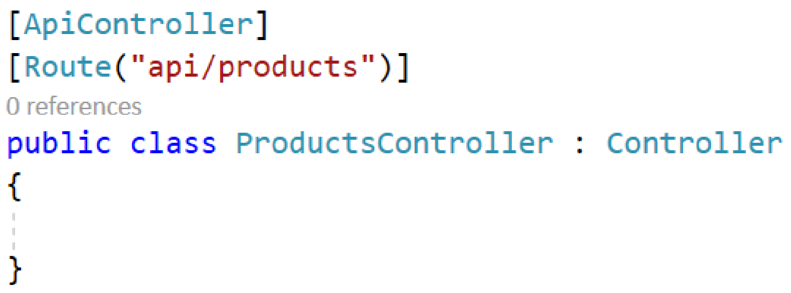
To **create an API controller**, you should first **create a standard controller class** and **add the needed attributes**. Create the **`ProductsController` class** in the "**`Controllers`**" **folder**:



```
Solution 'ProductsApi' (1 of 1 project)
  ProductsApi
    Connected Services
    Dependencies
    Properties
    Controllers
      ProductsController.cs
```

```
public class ProductsController : Controller
{
    0 references
    public IActionResult Index()
    {
        return View();
    }
}
```

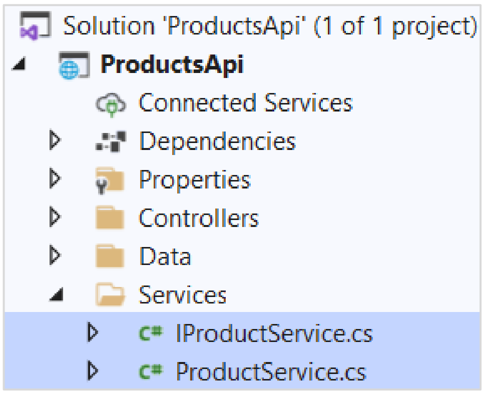
Remove the `Index()` method, as we won't need it. Then, add the **`[ApiController]`** and **`[Route]` attributes** to make the controller and **API controller**:



```
[ApiController]
[Route("api/products")]
0 references
public class ProductsController : 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**:



```
public interface IProductService
{
}
```

```
public class ProductService : IProductService
{
    private readonly ProductDbContext data;

    0 references
    public ProductService(ProductDbContext data)
        => this.data = data;
}
```

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

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

```
[ApiController]
[Route("api/[controller]")]
1 reference
public class ProductsController : Controller
{
    private readonly IProductService productService;

    0 references
    public ProductsController(IProductService productService)
        => this.productService = productService;
}
```

Step 4: Write the API Controller Methods

GetProducts() Method

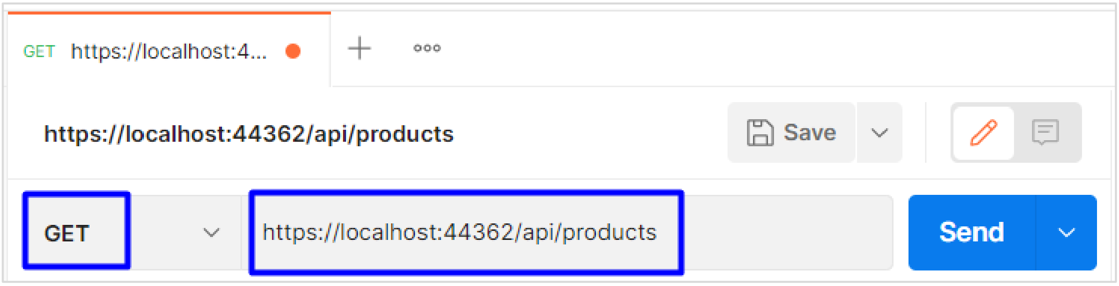
The first method should **return all products** as an **`ActionResult`** with a **collection of type `Product`** (we won't create and return a model, as we have a pretty basic class for the product). The **controller method** should use a **service method** and should be **invoked on a "GET" request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

```
[ApiController]
[Route("api/[controller]")]
1 reference
public class ProductsController : Controller
{
    ...
    [HttpGet]
    0 references
    public ActionResult<IEnumerable<Product>> GetProducts()
    {
        return this.productService.GetAllProducts();
    }
```

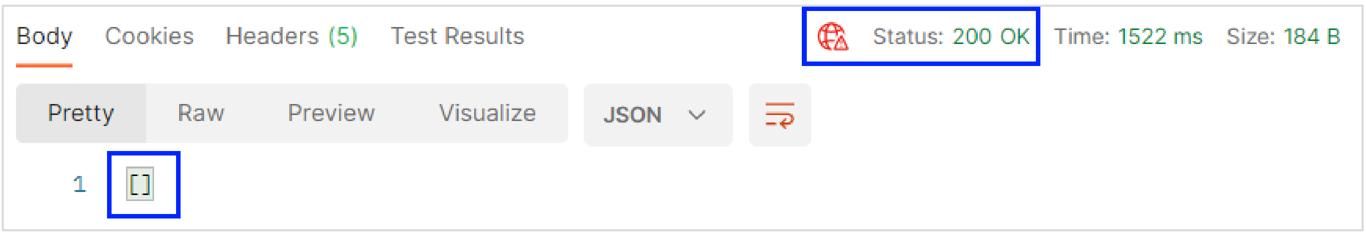
The **`IProductService`** and **`ProductService` classes** define and implement the **`GetAllProducts()` method**:

```
public interface IProductService
{
    List<Product> GetAllProducts();


public class ProductService : IProductService
{
    public List<Product> GetAllProducts()
        => this.data.Products.ToList();
```

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

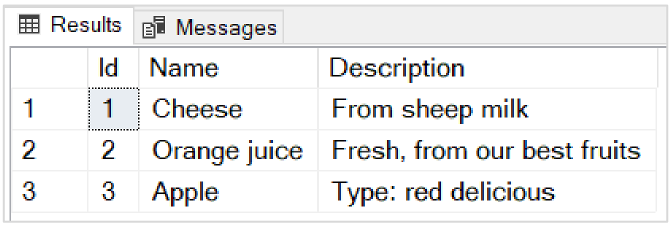

GET https://localhost:4...
https://localhost:44362/api/products
Save
GET
https://localhost:44362/api/products
Send

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:



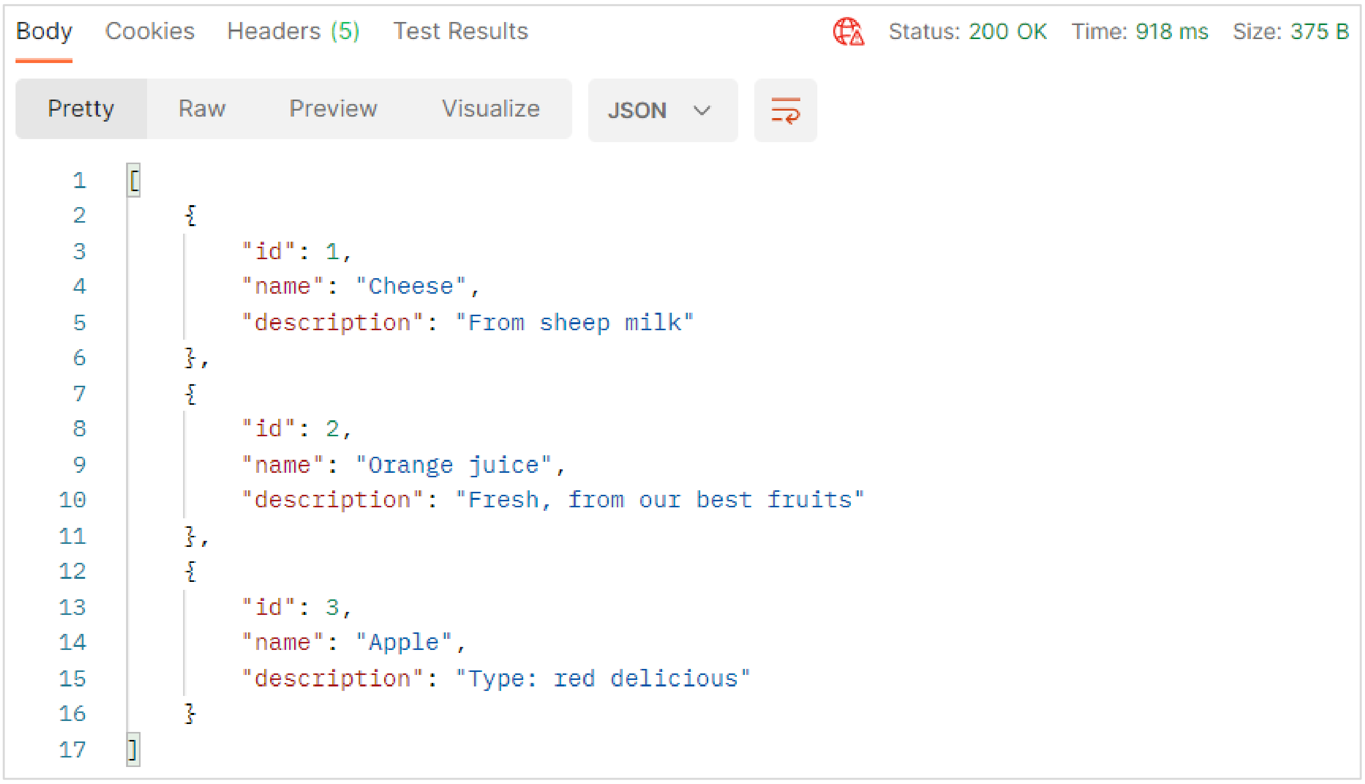
```
[]
```

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


Results
Messages

	Id	Name	Description
1	1	Cheese	From sheep milk
2	2	Orange juice	Fresh, from our best fruits
3	3	Apple	Type: red delicious

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



```
[
    {
        "id": 1,
        "name": "Cheese",
        "description": "From sheep milk"
    },
    {
        "id": 2,
        "name": "Orange juice",
        "description": "Fresh, from our best fruits"
    },
    {
        "id": 3,
        "name": "Apple",
        "description": "Type: red delicious"
    }
]
```

GetProduct() Method

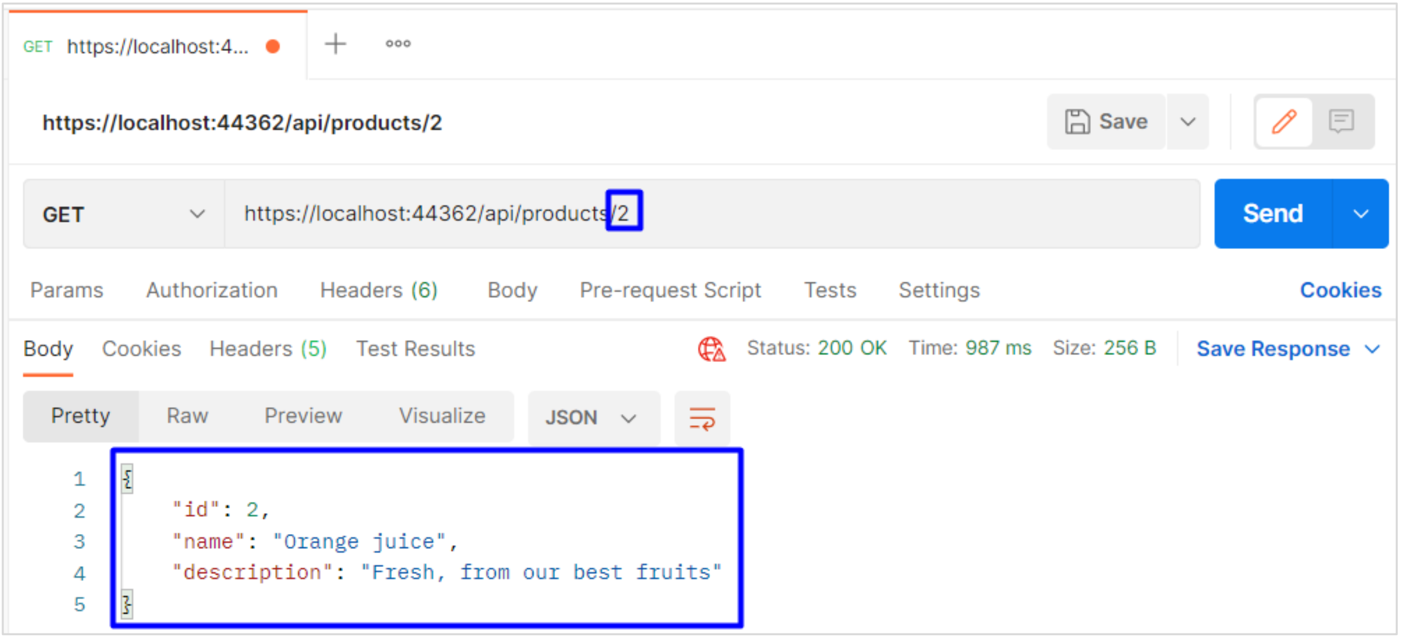
The **GetProduct(int id)** method should **return a product by a given id** if it exists. If it **doesn't exist**, a "**404 Not Found**" **response** should be returned. The method should be **invoked on a "GET" request to "/api/products/{id}"**. Write it in the **ProductsController class** like this:

```
[ApiController]
[Route("api/[controller]")]
1 reference
public class ProductsController : Controller
{
    ...
    [HttpGet("{id}")]
    0 references
    public ActionResult<Product> GetProduct(int id)
    {
        var product = this.productService.GetById(id);

        if (product == null) return NotFound();

        return product;
    }
```

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

```
public interface IProductService
{
    ...
    Product GetById(int id);
public class ProductService : IProductService
{
    ...
    public Product GetById(int id)
        => this.data.Products.Find(id);
```

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**:



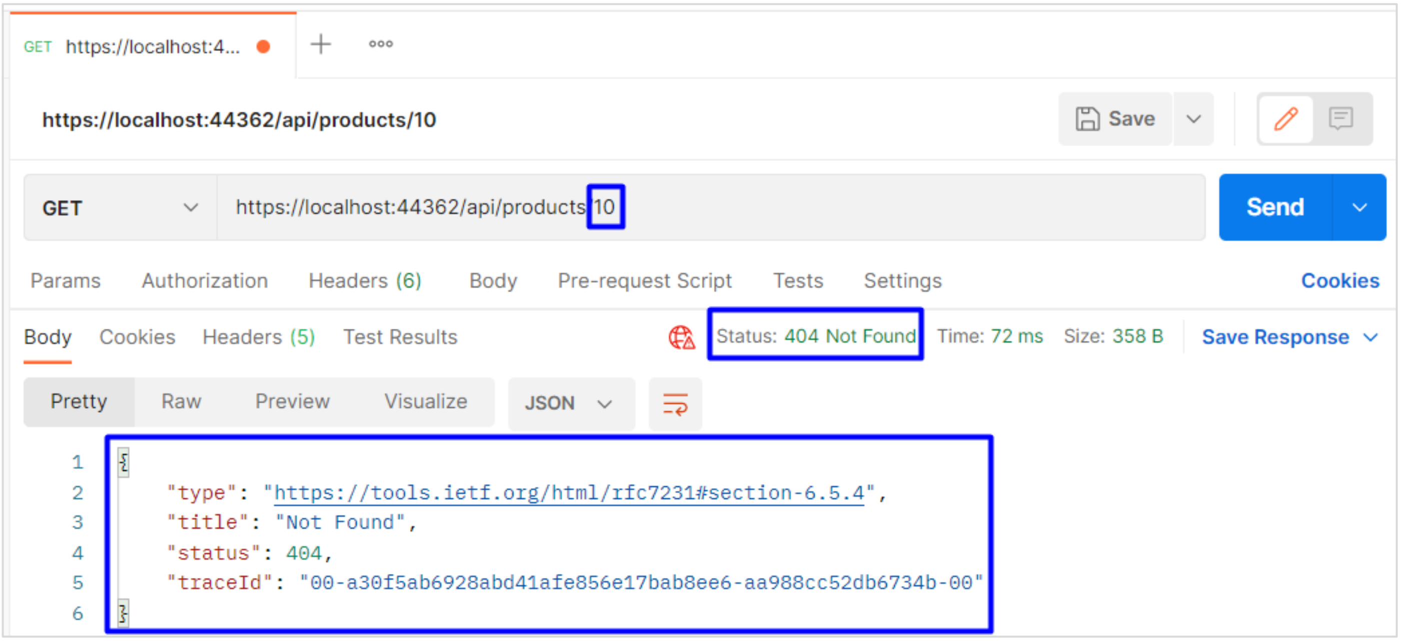
https://localhost:44362/api/products/2

GET https://localhost:44362/api/products/2

Status: 200 OK Time: 987 ms Size: 256 B

```
{
    "id": 2,
    "name": "Orange juice",
    "description": "Fresh, from our best fruits"
}
```

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



https://localhost:44362/api/products/10

GET https://localhost:44362/api/products/10

Status: 404 Not Found Time: 72 ms Size: 358 B

```
{
    "type": "https://tools.ietf.org/html/rfc7231#section-6.5.4",
    "title": "Not Found",
    "status": 404,
    "traceId": "00-a30f5ab6928abd41afe856e17bab8ee6-aa988cc52db6734b-00"
}
```

PostProduct() Method

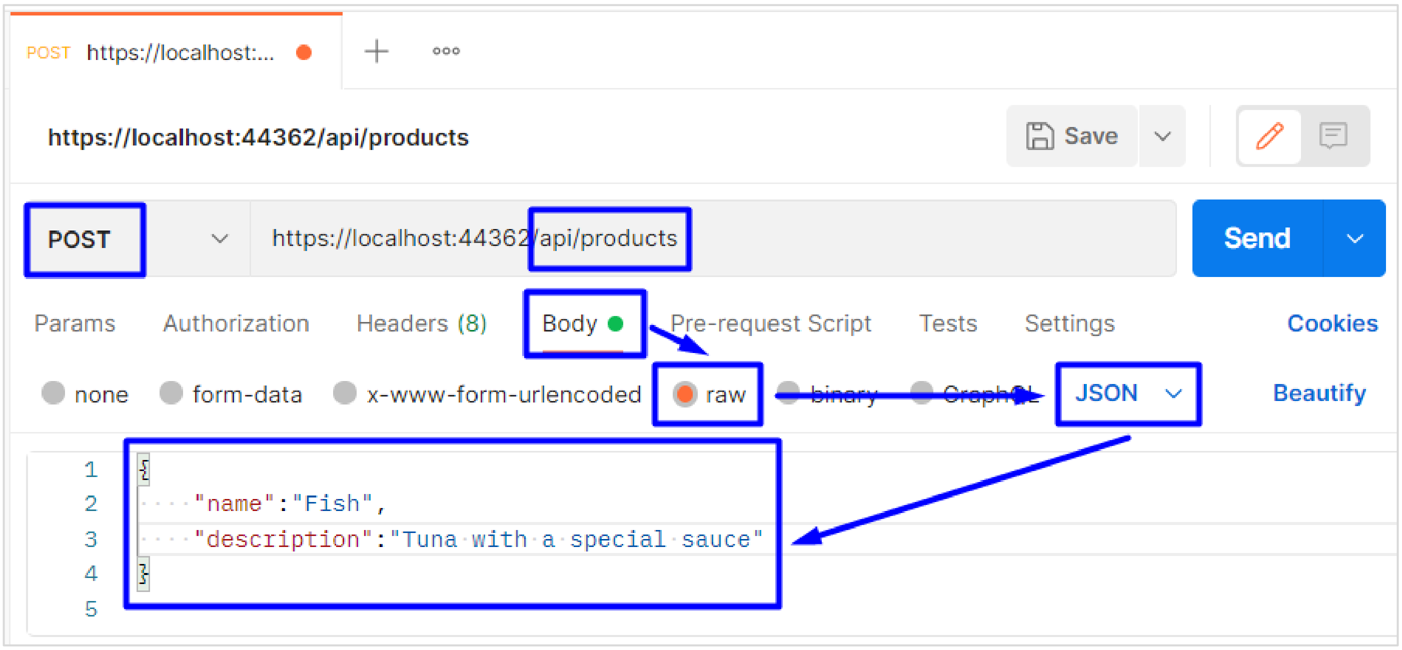
The **`PostProduct(Product product)` controller method** is responsible for **creating a new product** in the database. When the **product is created**, a "**`201 Created`**" **response** should be returned, which will invoke the **`GetProduct(int id)` method** to **return the product**. It should be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

```
[ApiController]
[Route("api/[controller]")]
1 reference
public class ProductsController : Controller
{
    ...
    [HttpPost]
    0 references
    public ActionResult<Product> PostProduct(Product product)
    {
        product = this.productService
            .CreateProduct(product.Name, product.Description);

        return CreatedAtAction("GetProduct", product);
    }
```

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

```
public interface IProductService
{
    ...
    Product CreateProduct
        (string name, string description);


public class ProductService : IProductService
{
    ...
    public Product CreateProduct(string name, string description)
    {
        var product = new Product()
        {
            Name = name,
            Description = description
        };

        this.data.Products.Add(product);
        this.data.SaveChanges();

        return product;
    }
```

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:

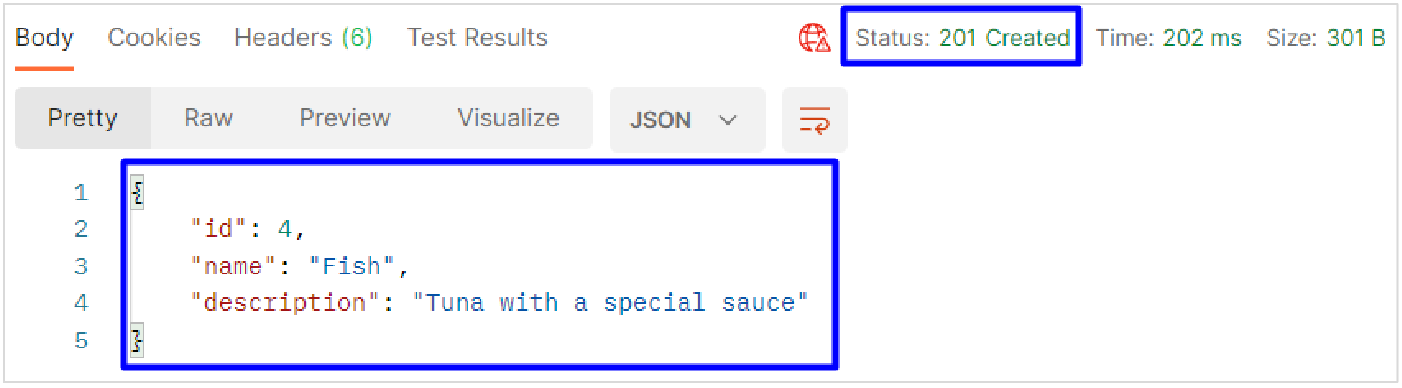


https://localhost:44362/api/products

POST https://localhost:44362/api/products

```
{
    "name":"Fish",
    "description":"Tuna with a special sauce"
}
```

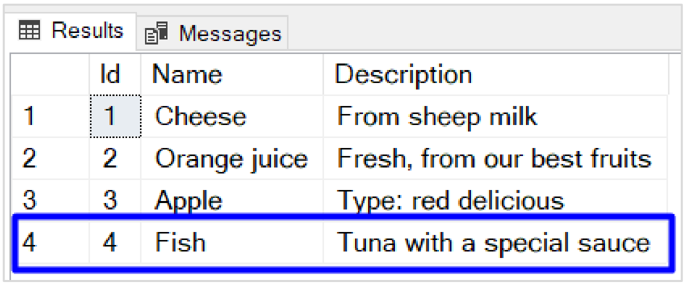
The **response** should be the following if the **product is created successfully**:



Status: 201 Created

```
{
    "id": 4,
    "name": "Fish",
    "description": "Tuna with a special sauce"
}
```

The **new product** should appear in the **database**:


Results
Messages

	Id	Name	Description
1	1	Cheese	From sheep milk
2	2	Orange juice	Fresh, from our best fruits
3	3	Apple	Type: red delicious
4	4	Fish	Tuna with a special sauce

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:

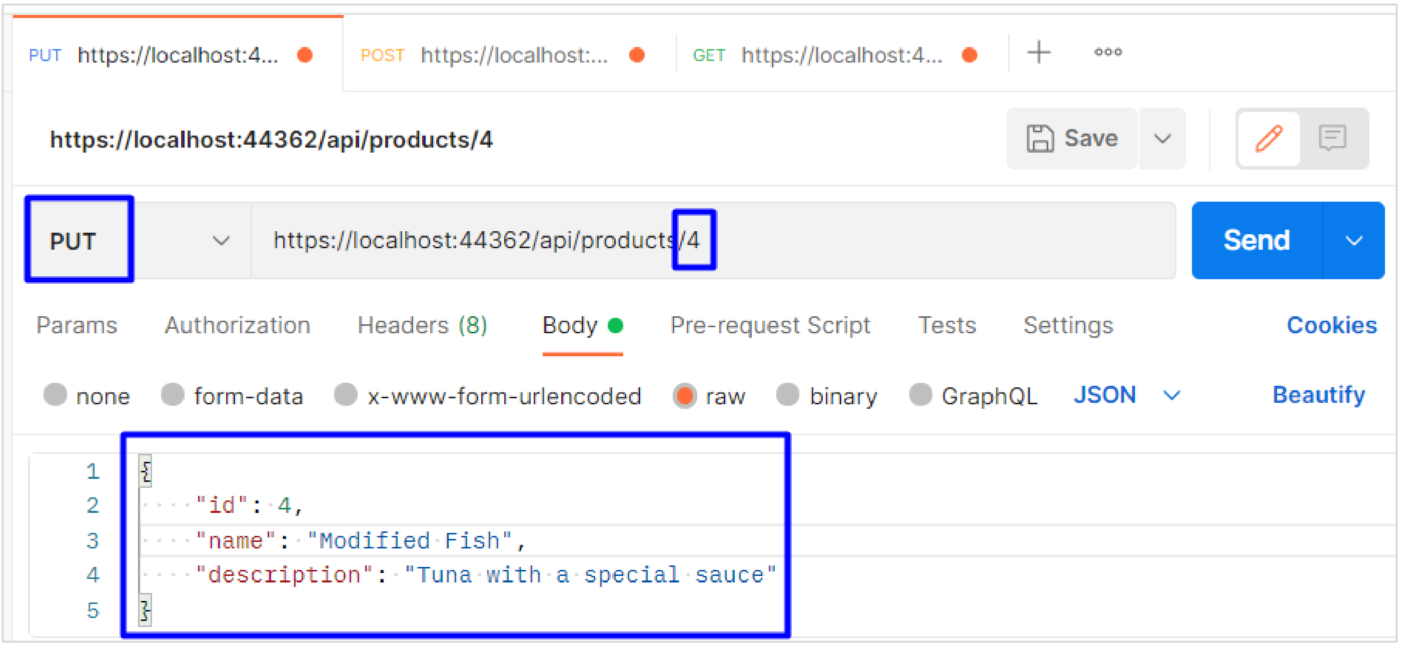
```
[ApiController]
[Route("api/[controller]")]
1 reference
public class ProductsController : Controller
{
    ...
    [HttpPut("{id}")]
    0 references
    public IActionResult PutProduct(int id, Product product)
    {
        if (id != product.Id) return BadRequest();

        if (this.productService.GetById(id) == null) return NotFound();

        this.productService.EditProduct(id, product);

        return NoContent();
    }
```

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

```
public interface IProductService
{
    ...
    void EditProduct(int id, Product product);


public class ProductService : IProductService
{
    ...
    public void EditProduct(int id, Product product)
    {
        var dbProduct = this.data.Products.Find(id);

        dbProduct.Name = product.Name;
        dbProduct.Description = product.Description;

        this.data.SaveChanges();
    }
```

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:

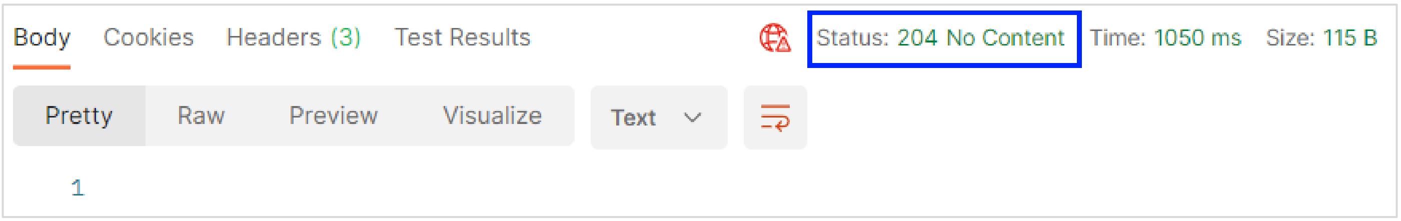


https://localhost:44362/api/products/4

PUT https://localhost:44362/api/products/4

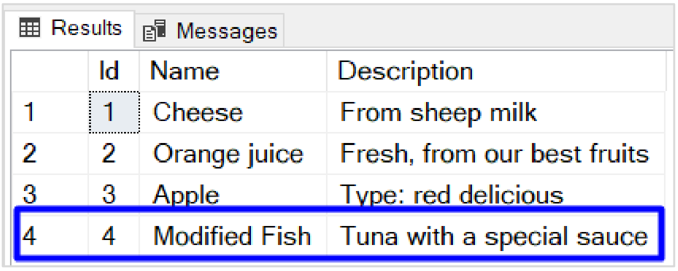
```
{
    "id": 4,
    "name": "Modified Fish",
    "description": "Tuna with a special sauce"
}
```

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



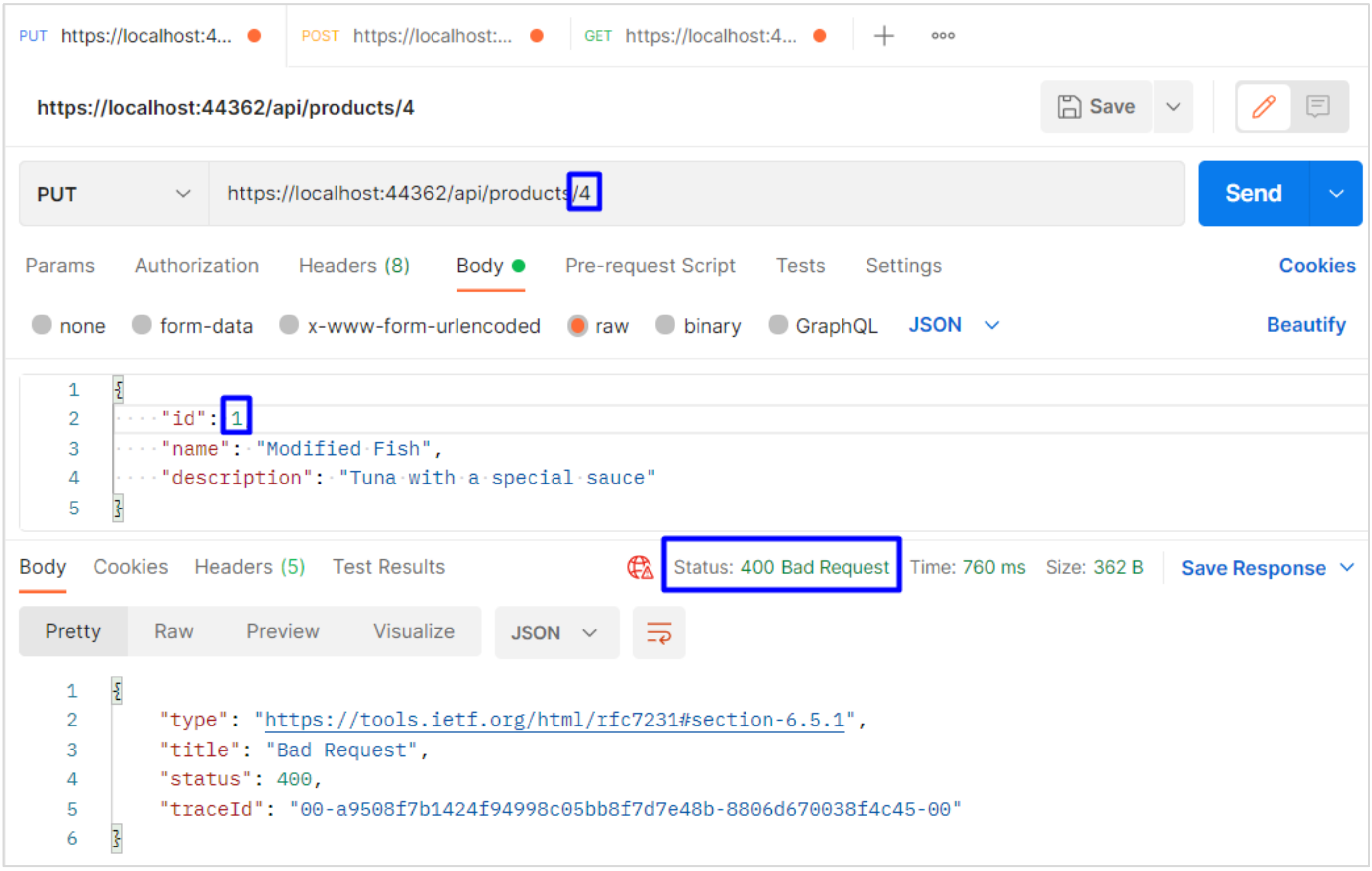
Status: 204 No Content Time: 1050 ms Size: 115 B

The **product** should be **modified in the database**, as well:



	Id	Name	Description
1	1	Cheese	From sheep milk
2	2	Orange juice	Fresh, from our best fruits
3	3	Apple	Type: red delicious
4	4	Modified Fish	Tuna with a special sauce

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



https://localhost:44362/api/products/4

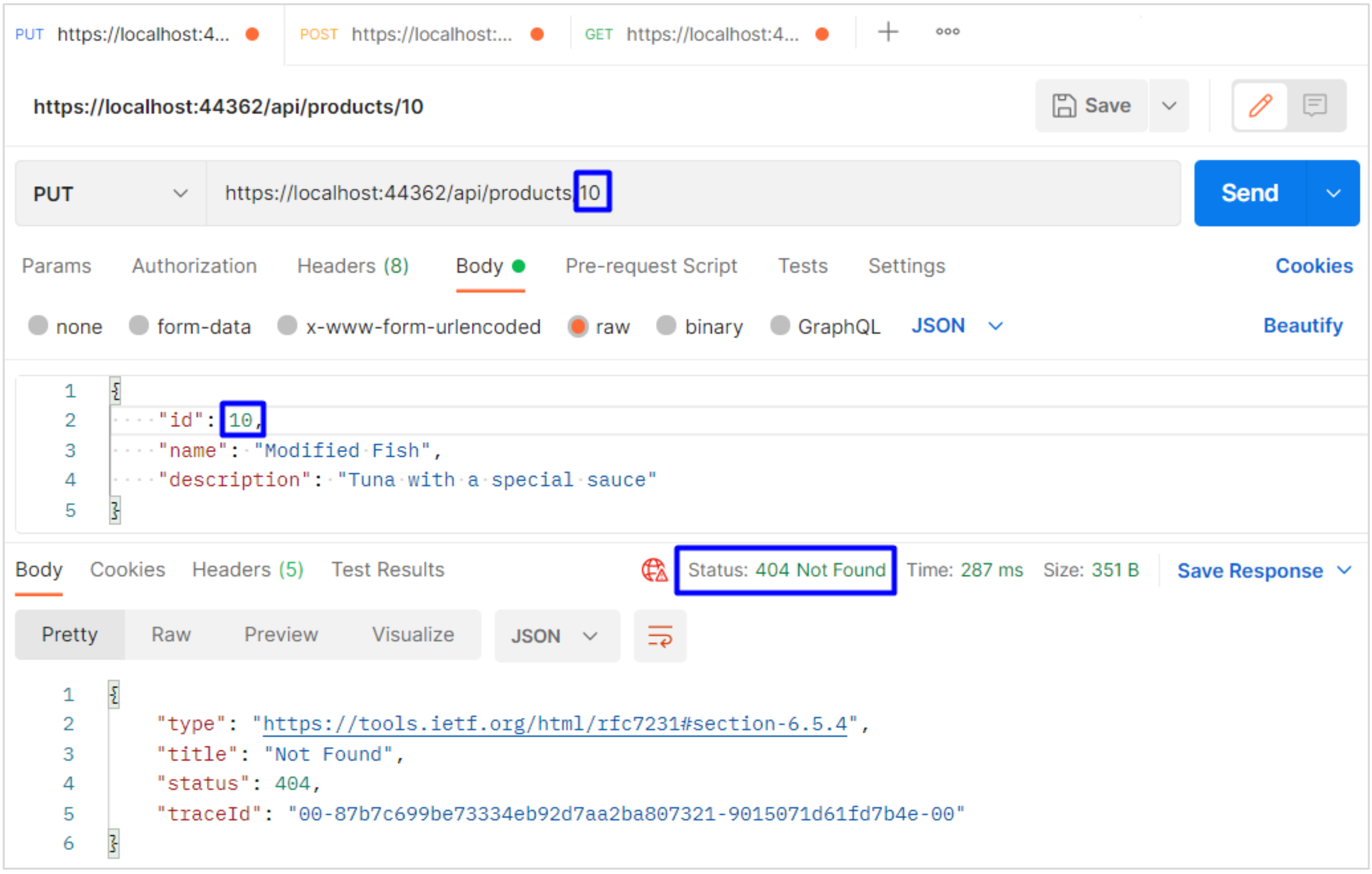
PUT https://localhost:44362/api/products/4

```
{
    "id": 1,
    "name": "Modified Fish",
    "description": "Tuna with a special sauce"
}
```

Status: 400 Bad Request Time: 760 ms Size: 362 B

```
{
    "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
    "title": "Bad Request",
    "status": 400,
    "traceId": "00-a9508f7b1424f94998c05bb8f7d7e48b-8806d670038f4c45-00"
}
```

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



https://localhost:44362/api/products/10

PUT https://localhost:44362/api/products/10

```
{
    "id": 10,
    "name": "Modified Fish",
    "description": "Tuna with a special sauce"
}
```

Status: 404 Not Found Time: 287 ms Size: 351 B

```
{
    "type": "https://tools.ietf.org/html/rfc7231#section-6.5.4",
    "title": "Not Found",
    "status": 404,
    "traceId": "00-87b7c699be73334eb92d7aa2ba807321-9015071d61fd7b4e-00"
}
```

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:

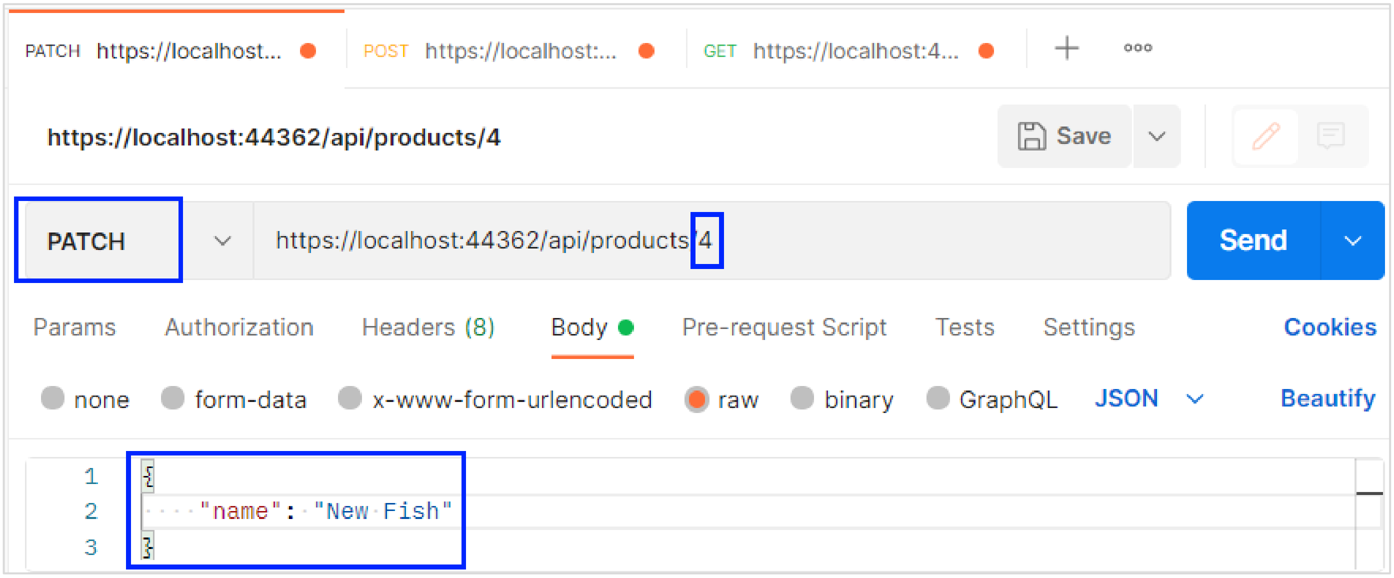
```
[ApiController]
[Route("api/[controller]")]
1 reference
public class ProductsController : Controller
{
    ...
    [HttpPatch("{id}")]
    0 references
    public IActionResult PatchProduct(int id, Product product)
    {
        if (this.productService.GetById(id) == null) return NotFound();

        this.productService.EditProductPartially(id, product);

        return NoContent();
    }
```

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

```
public interface IProductService
{
    ...
    2 references
    void EditProductPartially(int id, Product product);
```

```
public class ProductService : IProductService
{
    ...

    public void EditProductPartially(int id, Product product)
    {
        var dbProduct = this.data.Products.Find(id);

        dbProduct.Name = String.IsNullOrEmpty(product.Name)
            ? dbProduct.Name : product.Name;
        dbProduct.Description = String.IsNullOrEmpty(product.Description)
            ? dbProduct.Description : product.Description;

        this.data.SaveChanges();
    }
```

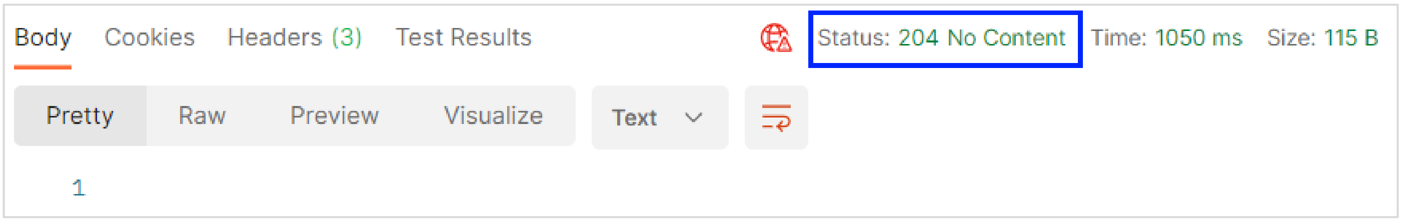
Now create a "**PATCH**" **request** in **Postman** to "**`/api/products/{id}`**" with a **valid id** and with **modified data only**:



PATCH https://localhost:44362/api/products/4

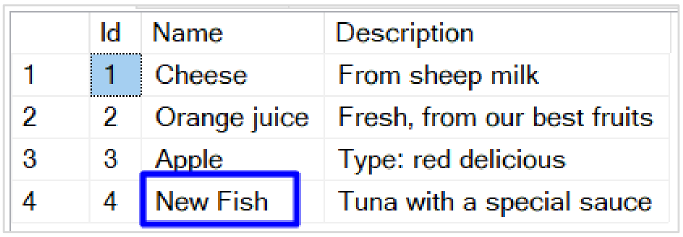
```
{
    "name": "New Fish"
}
```

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



Status: 204 No Content Time: 1050 ms Size: 115 B

Check the **modification in the database**, as well:



	Id	Name	Description
1	1	Cheese	From sheep milk
2	2	Orange juice	Fresh, from our best fruits
3	3	Apple	Type: red delicious
4	4	New Fish	Tuna with a special sauce

DeleteProduct() Method

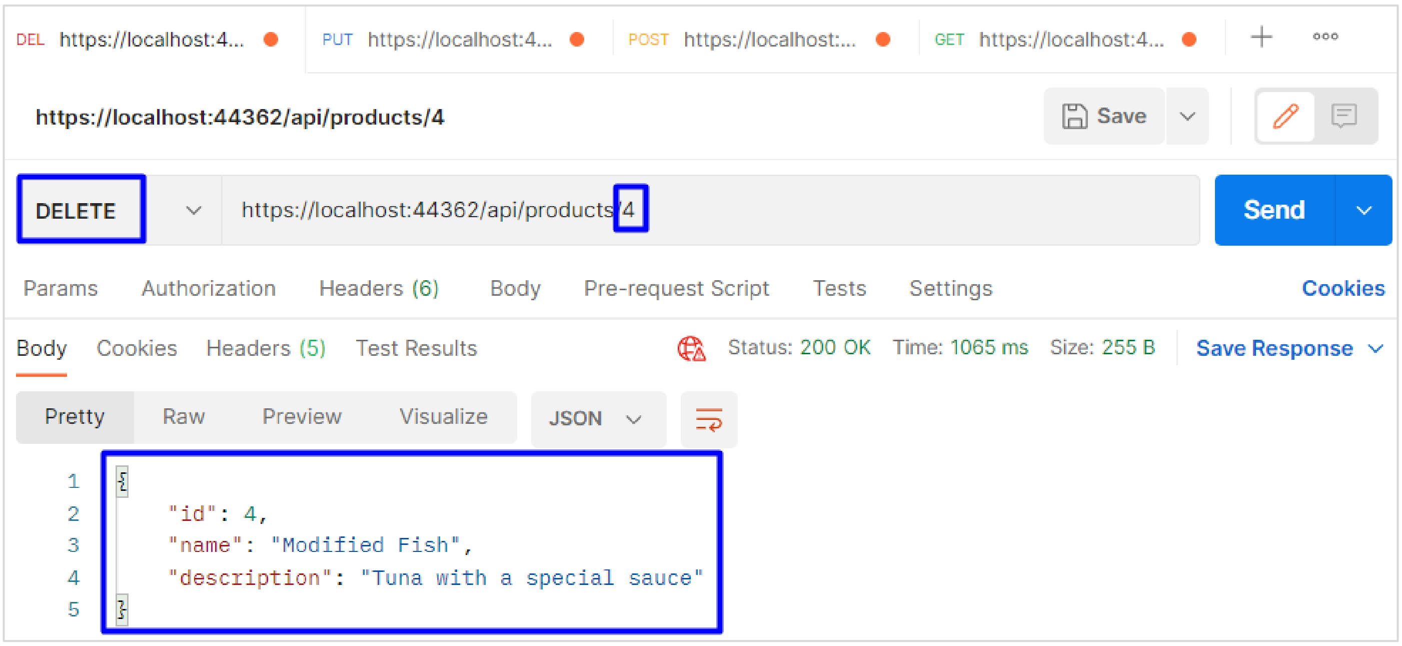
The **`DeleteProduct(int id)` method** is the last method we will implement. It should be invoked on a "**`DELETE`**" **request** to "**`/api/products/{id}`**". If a **product with the given id doesn't exist**, "**`404` Not Found**" is returned. If it **exists**, the **deleted product is returned**:

```
[ApiController]
[Route("api/[controller]")]
1 reference
public class ProductsController : Controller
{
    ...
    [HttpDelete("{id}")]
    0 references
    public ActionResult<Product> DeleteProduct(int id)
    {
        if (this.productService.GetById(id) == null) return NotFound();

        var product = this.productService.DeleteProduct(id);

        return product;
    }
```

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

```
public interface IProductService
{
    ...
    Product DeleteProduct(int id);


public class ProductService : IProductService
{
    ...
    public Product DeleteProduct(int id)
    {
        var product = this.data.Products.Find(id);

        this.data.Products.Remove(product);
        this.data.SaveChanges();

        return product;
    }
```

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



https://localhost:44362/api/products/4

DELETE https://localhost:44362/api/products/4

Status: 200 OK Time: 1065 ms Size: 255 B

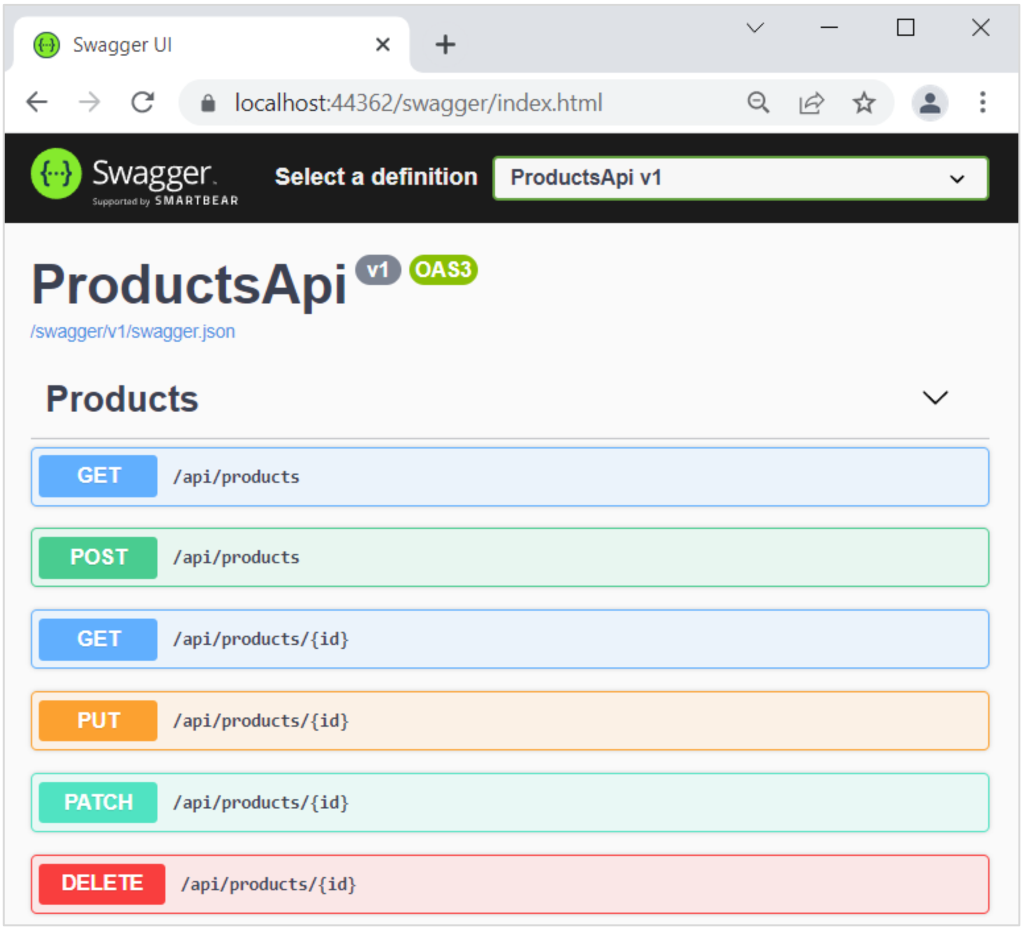
```
{
    "id": 4,
    "name": "Modified Fish",
    "description": "Tuna with a special sauce"
}
```

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**:



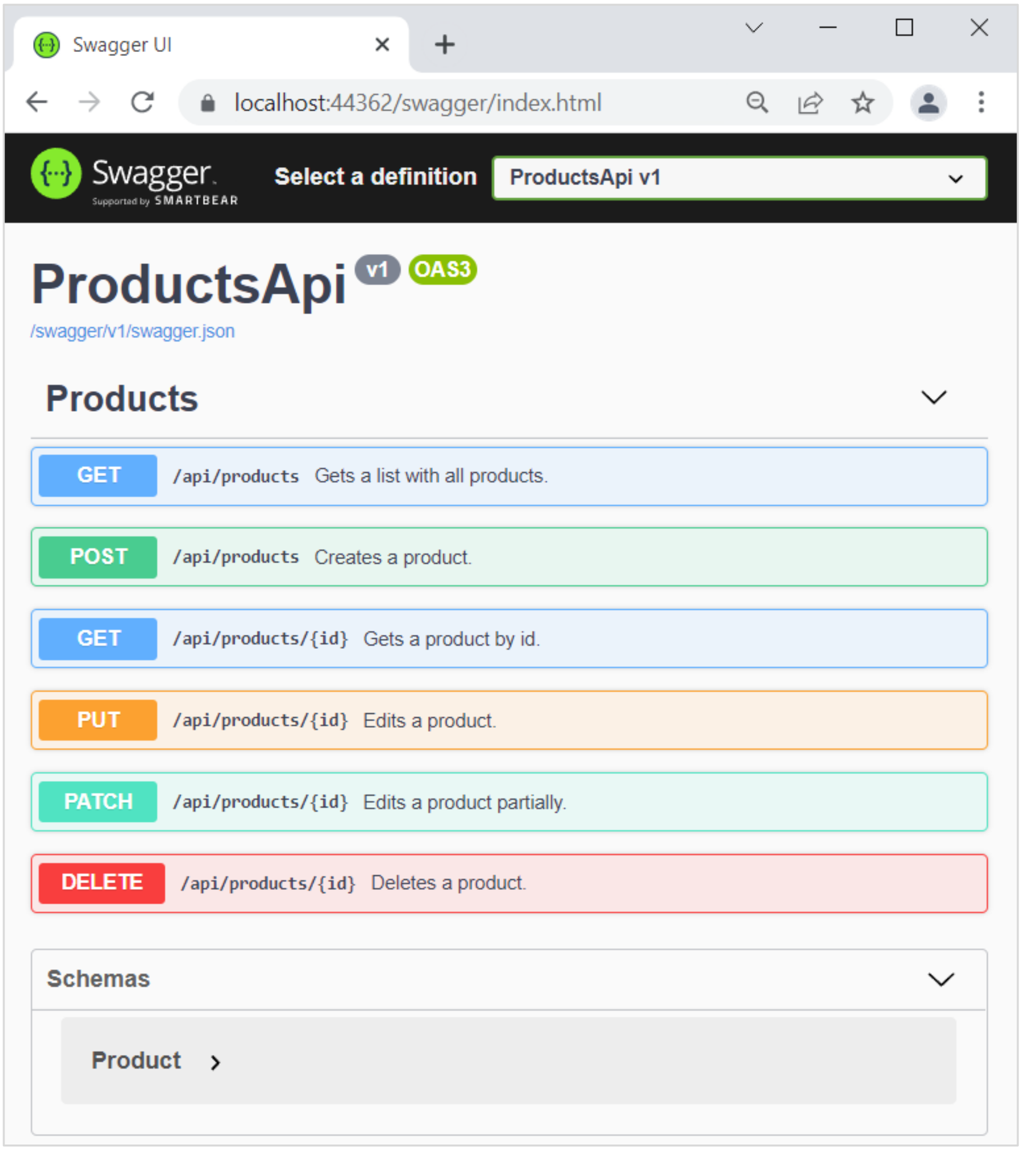
localhost:44362/swagger/index.html

ProductsApi

/swagger/v1/swagger.json

```
GET     /api/products
POST    /api/products
GET     /api/products/{id}
PUT     /api/products/{id}
PATCH   /api/products/{id}
DELETE  /api/products/{id}
```

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



ProductsApi v1 OAS3
/swagger/v1/swagger.json

Products

GET /api/products Gets a list with all products.
POST /api/products Creates a product.
GET /api/products/{id} Gets a product by id.
PUT /api/products/{id} Edits a product.
PATCH /api/products/{id} Edits a product partially.
DELETE /api/products/{id} Deletes a product.

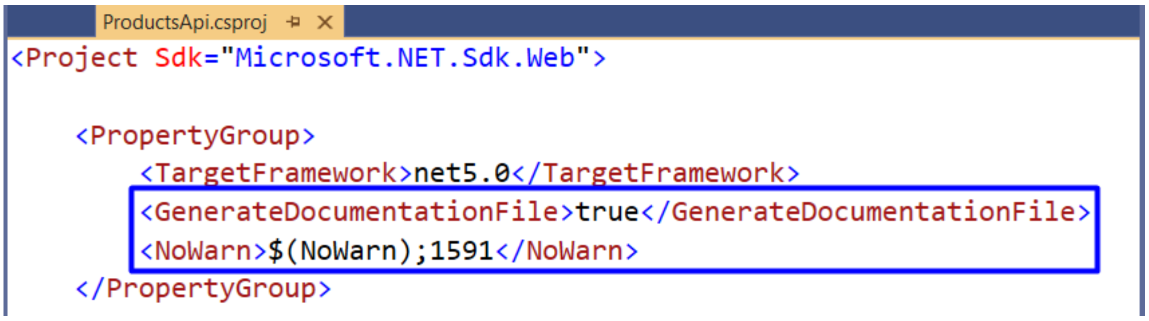
Schemas

Product >

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**:

```
public class Startup
{
    ...
    public void ConfigureServices(IServiceCollection services)
    {
        ...
        services.AddSwaggerGen(c =>
        {
            c.SwaggerDoc("v1", new OpenApiInfo
                { Title = "ProductsApi", Version = "v1" });

            var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
            var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
            c.IncludeXmlComments(xmlPath);
        });
    }
```

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**:

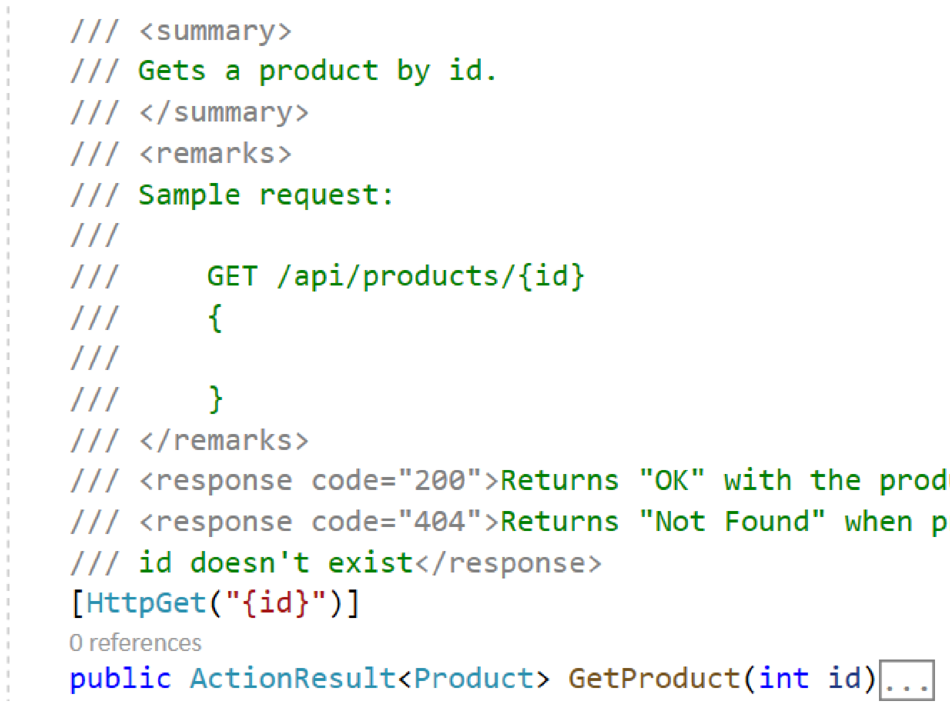


ProductsApi.csproj

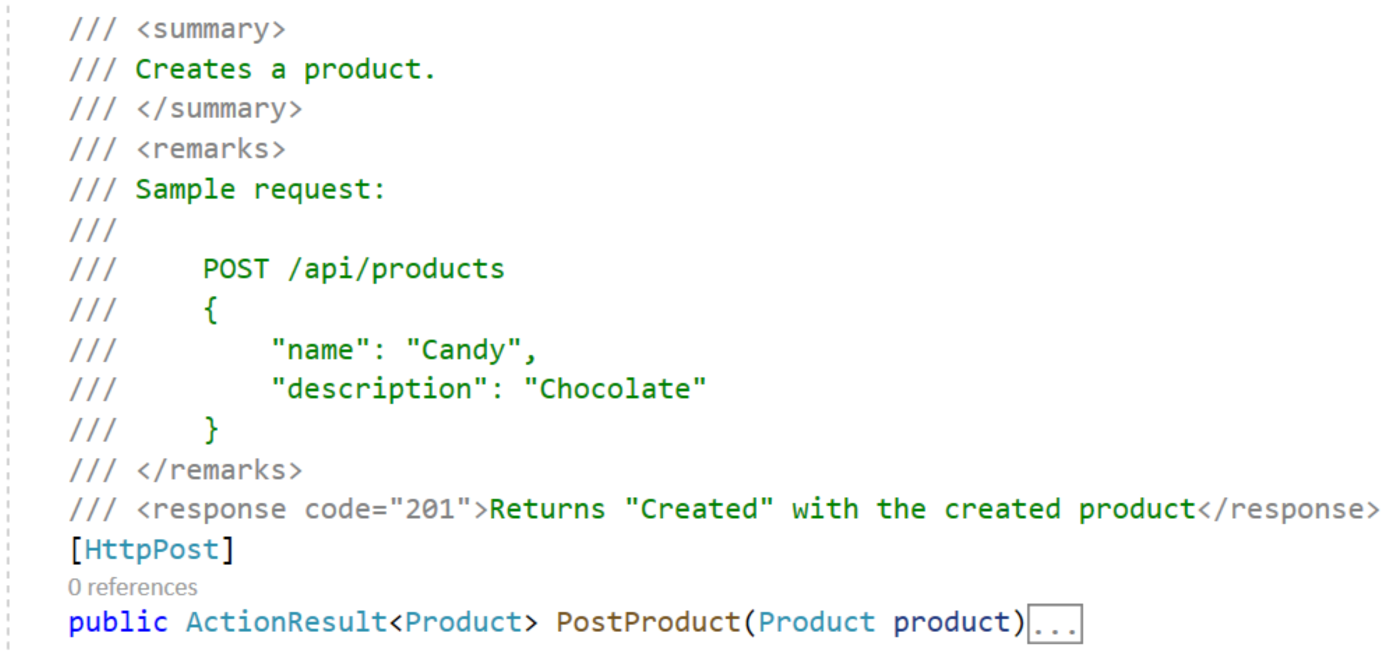
```
<Project Sdk="Microsoft.NET.Sdk.Web">

    <PropertyGroup>
        <TargetFramework>net5.0</TargetFramework>
        <GenerateDocumentationFile>true</GenerateDocumentationFile>
        <NoWarn>$(NoWarn);1591</NoWarn>
    </PropertyGroup>
```

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:

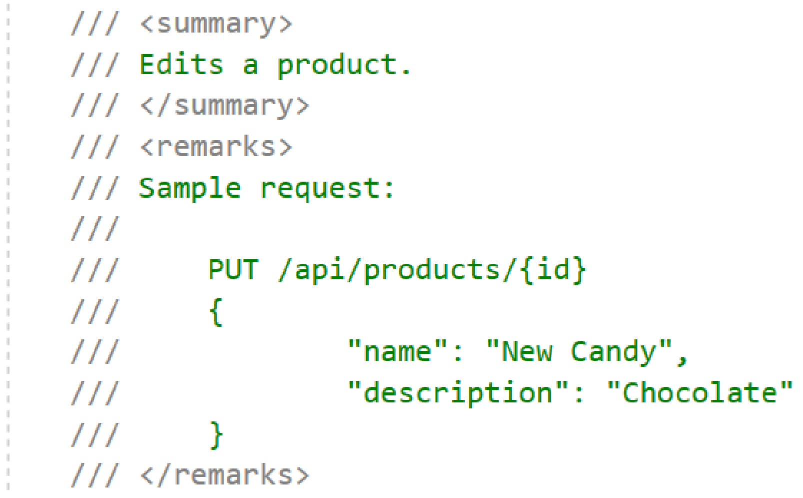
```
[ApiController]
[Route("api/products")]
1 reference
public class ProductsController : Controller
{
    private readonly IProductService productService;

    0 references
    public ProductsController(IProductService productService)
        => this.productService = productService;

    /// <summary>
    /// Gets a list with all products.
    /// </summary>
    /// <remarks>
    /// Sample request:
    ///
    ///     GET /api/products
    ///     {
    ///
    ///     }
    /// </remarks>
    /// <response code="200">Returns "OK" with a list of all products</response>
    [HttpGet]
    0 references
    public ActionResult<IEnumerable<Product>> GetProducts()...
```



```
/// <summary>
/// Gets a product by id.
/// </summary>
/// <remarks>
/// Sample request:
///
///     GET /api/products/{id}
///     {
///
///     }
/// </remarks>
/// <response code="200">Returns "OK" with the product</response>
/// <response code="404">Returns "Not Found" when product with the given
/// id doesn't exist</response>
[HttpGet("{id}")]
0 references
public ActionResult<Product> GetProduct(int id)...
```



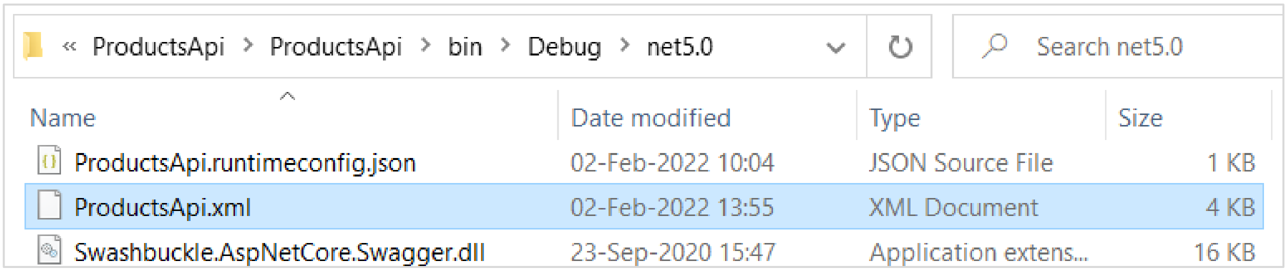
```
/// <summary>
/// Creates a product.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /api/products
///     {
///         "name": "Candy",
///         "description": "Chocolate"
///     }
/// </remarks>
/// <response code="201">Returns "Created" with the created product</response>
[HttpPost]
0 references
public ActionResult<Product> PostProduct(Product product)...
```



```
/// <summary>
/// Edits a product.
/// </summary>
/// <remarks>
/// Sample request:
///
///     PUT /api/products/{id}
///     {
///             "name": "New Candy",
///             "description": "Chocolate"
///     }
/// </remarks>
```

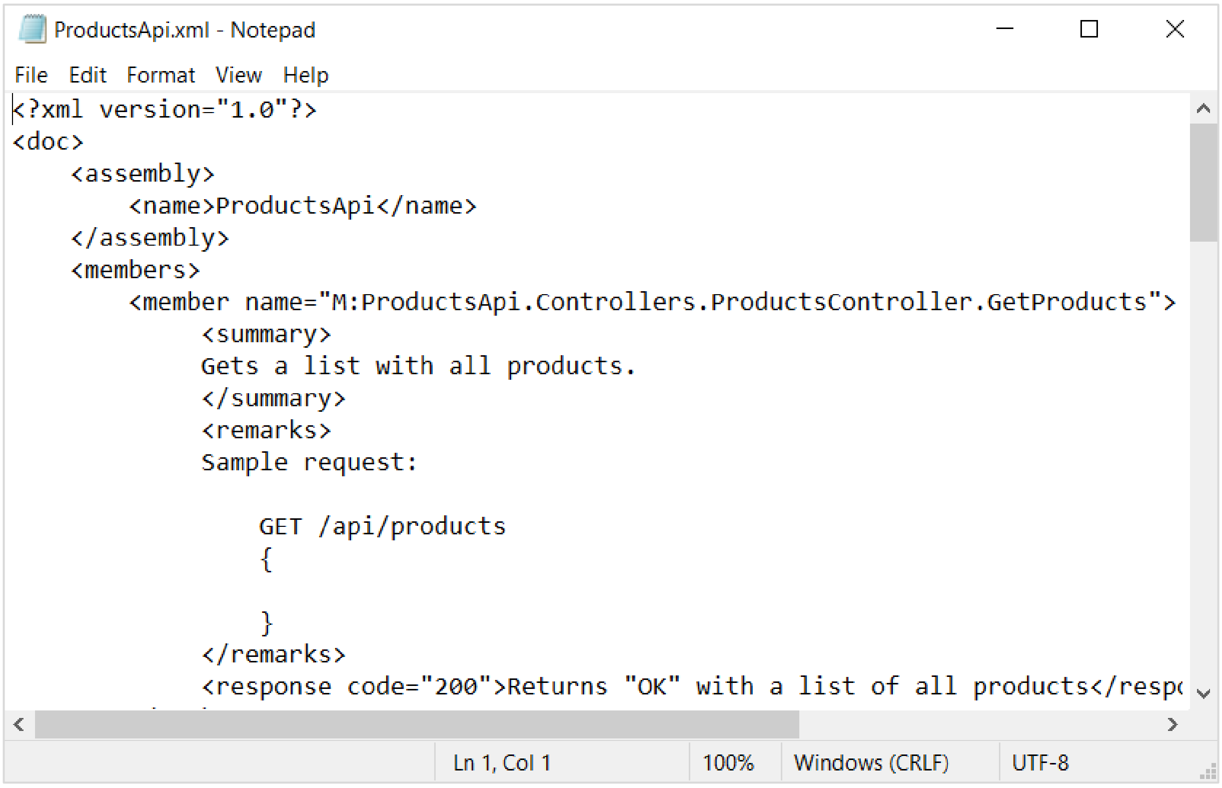
```
    /// <response code="204">Returns "No Content"</response>
    /// <response code="400">Returns "Bad Request" when an invalid
    /// request is sent</response>
    /// <response code="404">Returns "Not Found" when product with
    /// the given id doesn't exist</response>
    [HttpPut("{id}")]
    0 references
    public IActionResult PutProduct(int id, Product product)...

    /// <summary>
    /// Edits a product partially.
    /// </summary>
    /// <remarks>
    /// Sample request:
    ///
    ///     PUT /api/products/{id}
    ///     {
    ///             "name": "New Candy"
    ///     }
    /// </remarks>
    /// <response code="204">Returns "No Content"</response>
    /// <response code="404">Returns "Not Found" when product with
    /// the given id doesn't exist</response>
    [HttpPatch("{id}")]
    0 references
    public IActionResult PatchProduct(int id, Product product)...

    /// <summary>
    /// Deletes a product.
    /// </summary>
    /// <remarks>
    /// Sample request:
    ///
    ///     DELETE /api/products/{id}
    ///     {
    ///
    ///     }
    /// </remarks>
    /// <response code="200">Returns "OK" with the deleted product</response>
    /// <response code="404">Returns "Not Found" when product with the
    /// given id doesn't exist</response>
    [HttpDelete("{id}")]
    0 references
    public ActionResult<Product> DeleteProduct(int id)...
}
```

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**:



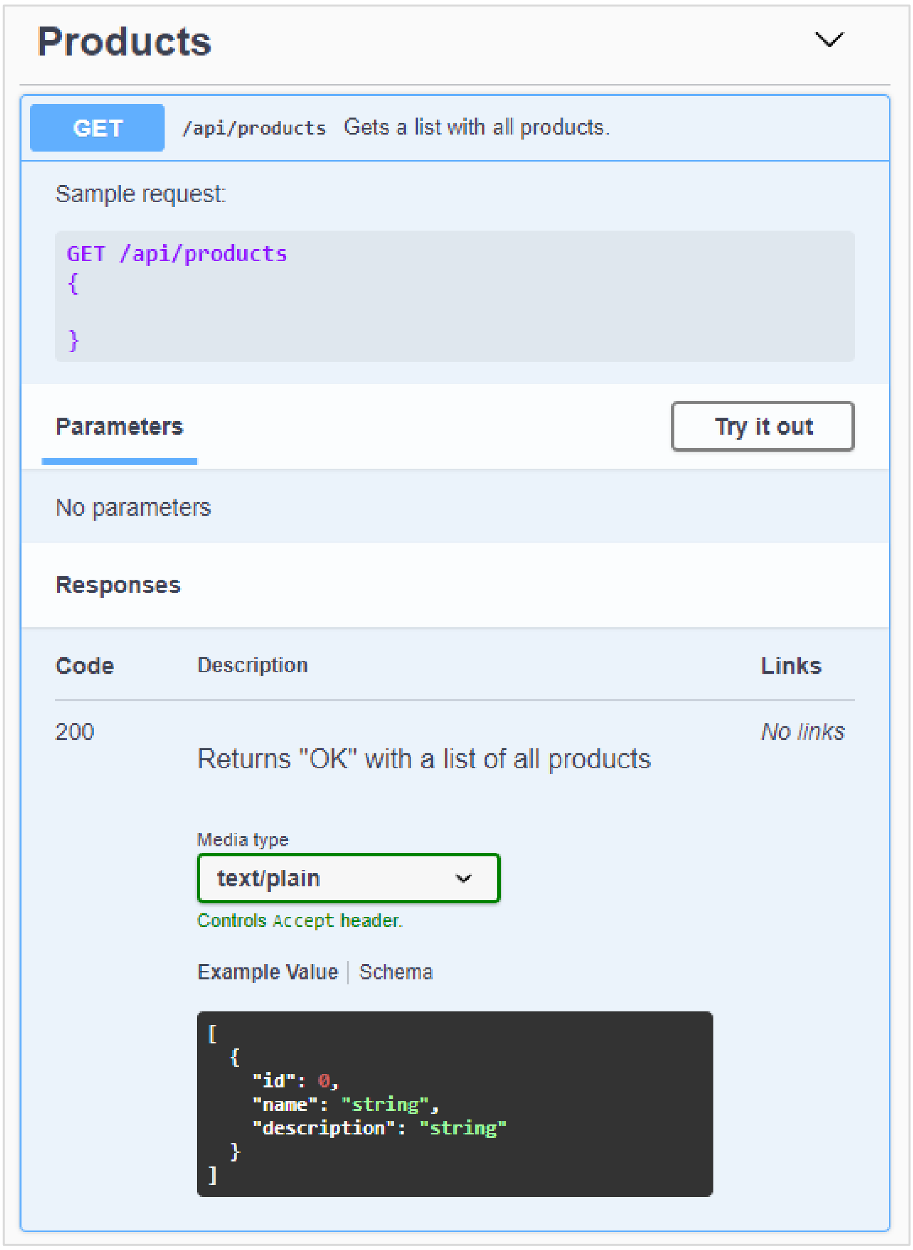
Name	Date modified	Type	Size
ProductsApi.runtimeconfig.json	02-Feb-2022 10:04	JSON Source File	1 KB
ProductsApi.xml	02-Feb-2022 13:55	XML Document	4 KB
Swashbuckle.AspNetCore.Swagger.dll	23-Sep-2020 15:47	Application extens...	16 KB

→



```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>ProductsApi</name>
    </assembly>
    <members>
        <member name="M:ProductsApi.Controllers.ProductsController.GetProducts">
             <summary>
             Gets a list with all products.
             </summary>
             <remarks>
             Sample request:

                 GET /api/products
                 {

                 }
             </remarks>
             <response code="200">Returns "OK" with a list of all products</respo
```

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



GET /api/products Gets a list with all products.

Sample request:

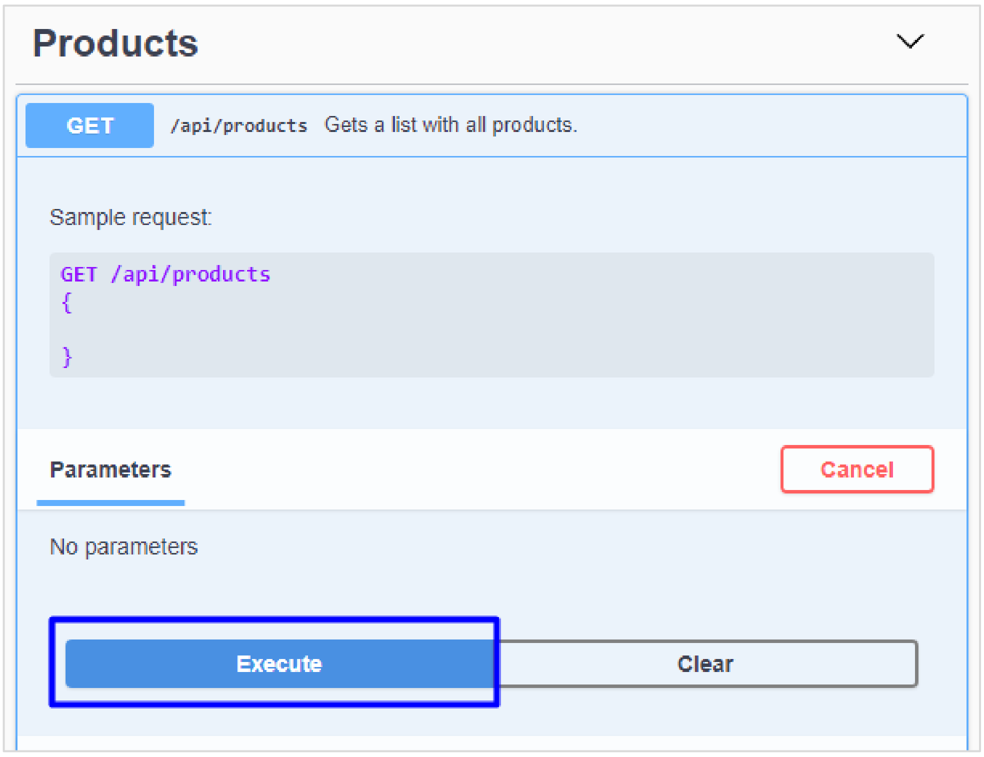
```
GET /api/products
{

}
```

No parameters

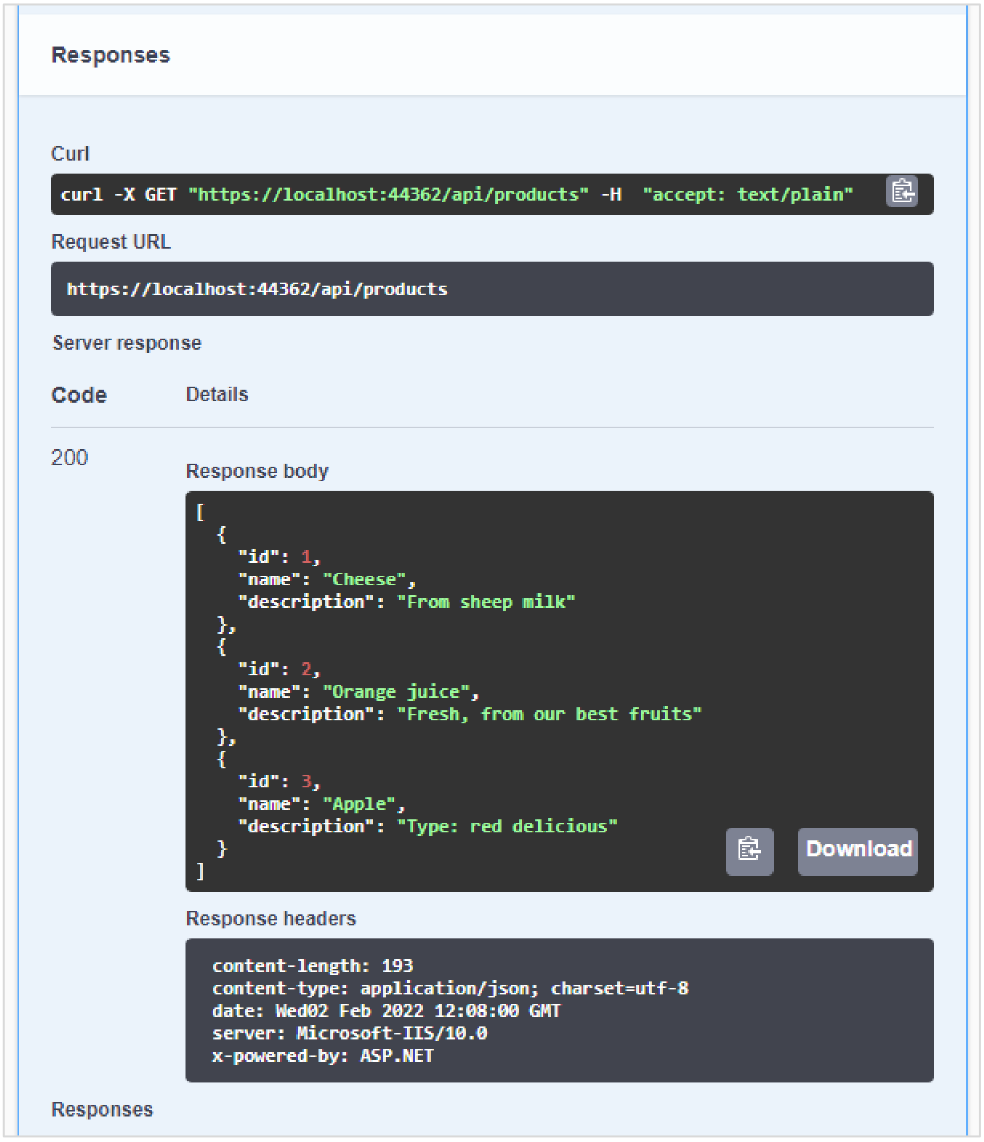
Code	Description	Links
200	Returns "OK" with a list of all products Media type: text/plain Controls Accept header.	*No links*

```
[
  {
    "id": 0,
    "name": "string",
    "description": "string"
  }
]
```

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**:



```
GET /api/products
{

}
```

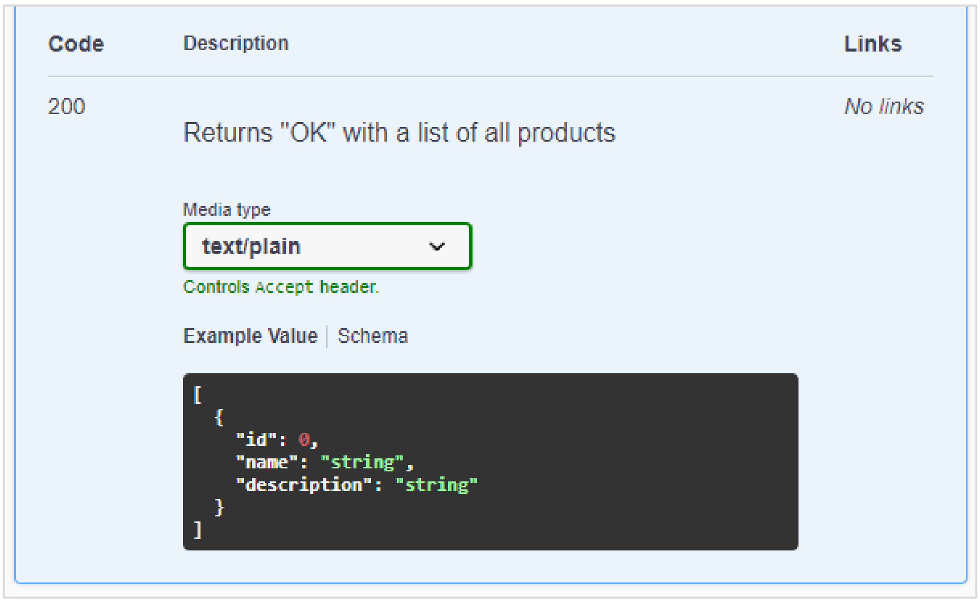


```
curl -X GET "https://localhost:44362/api/products" -H  "accept: text/plain"
```

```
https://localhost:44362/api/products
```

```
[
  {
    "id": 1,
    "name": "Cheese",
    "description": "From sheep milk"
  },
  {
    "id": 2,
    "name": "Orange juice",
    "description": "Fresh, from our best fruits"
  },
  {
    "id": 3,
    "name": "Apple",
    "description": "Type: red delicious"
  }
]
```

```
content-length: 193
content-type: application/json; charset=utf-8
date: Wed02 Feb 2022 12:08:00 GMT
server: Microsoft-IIS/10.0
x-powered-by: ASP.NET
```



Code	Description	Links
200	Returns "OK" with a list of all products	No links

Media type: text/plain

```
[
  {
    "id": 0,
    "name": "string",
    "description": "string"
  }
]
```

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