

# Implementing .NET Core Web API with Entity Framework

*Sanduni Sasanthika*



Photo by [Burst](#) on [Unsplash](#)

Welcome to the world of web development! In this guide, we'll take a straightforward approach to building a .NET Core Web API with Entity Framework. No jargon, just easy steps to get you started. If you're aiming to create APIs that play well with modern applications, you're in the right place. We'll be using .NET Core, Entity Framework 8, and Microsoft SQL Server to make the process smooth and efficient.

Whether you're a coding pro or just starting, our goal is to guide you through the process of setting up a powerful API. We'll focus on CRUD operations (Create, Read, Update, Delete) and take a Database-First approach.

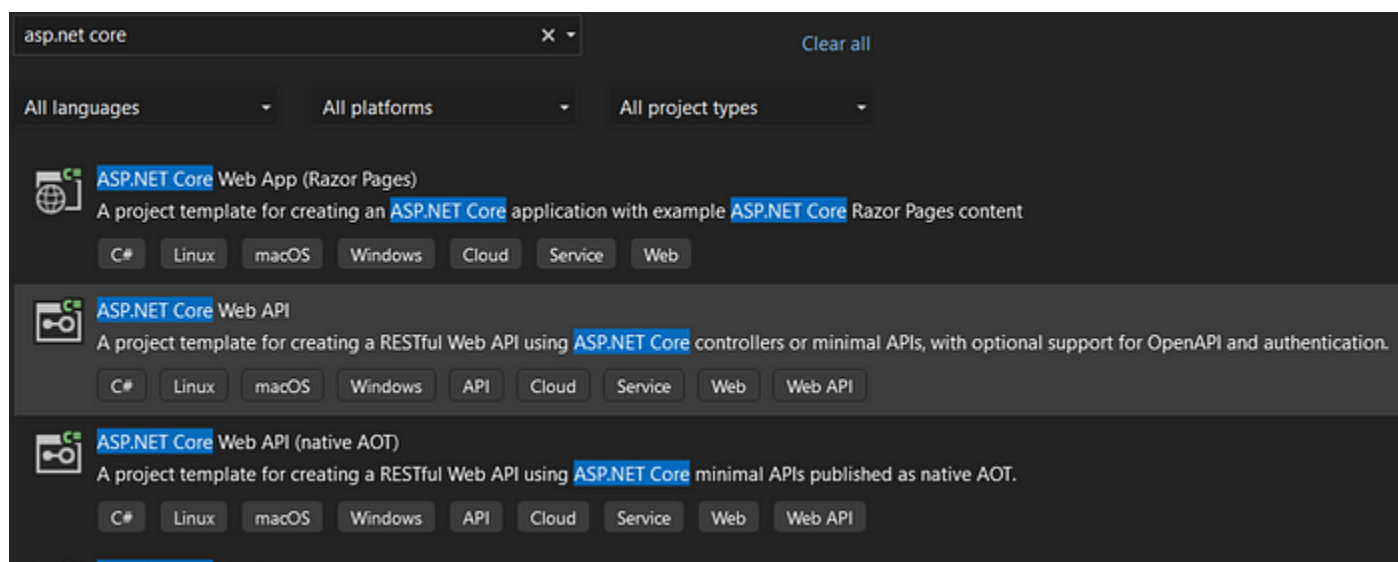
So, if you're ready to roll up your sleeves and dive into the world of .NET Core, let's get started on building a Web API that's simple, effective, and ready for action.

*(The following project is done as a team effort.)*

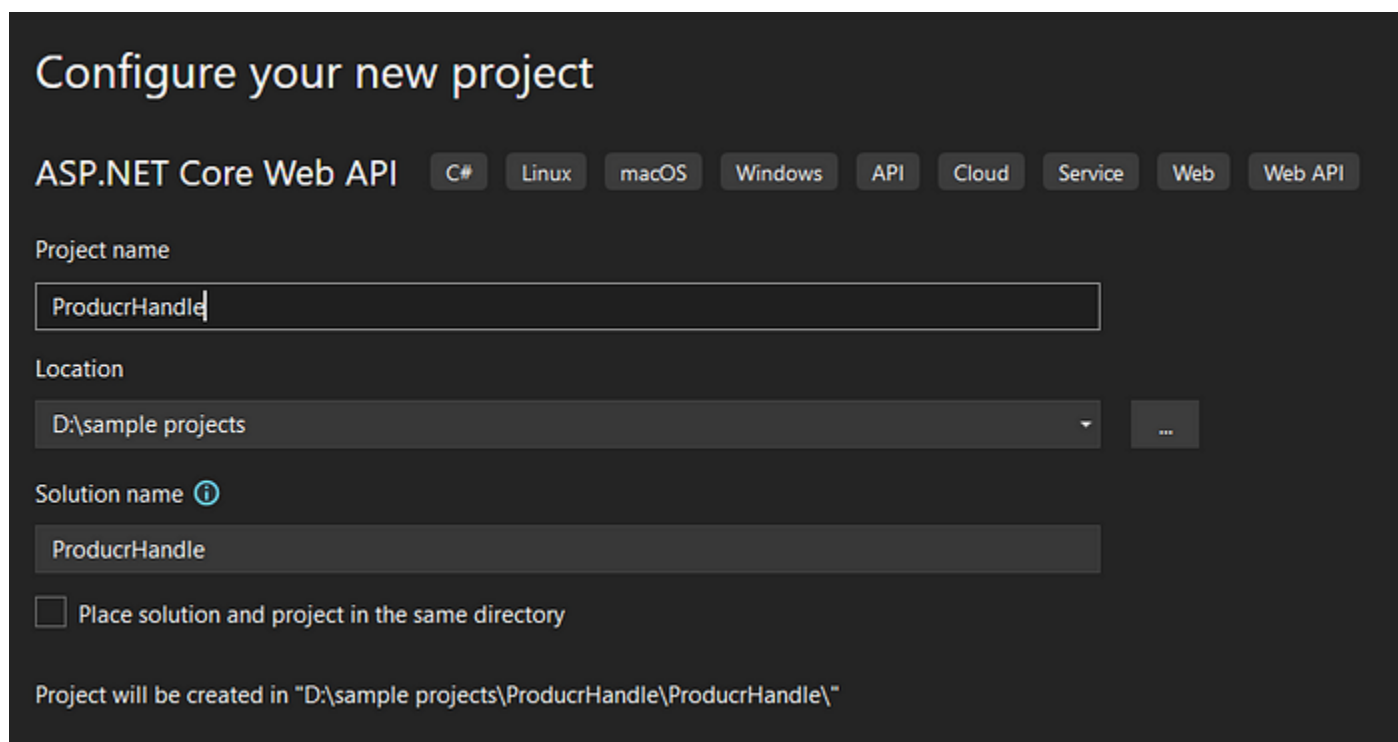
## Create Web API using Visual Studio 2022

Select "Create a New Project"

Choose the **ASP.NET Core Web API** as the project template.

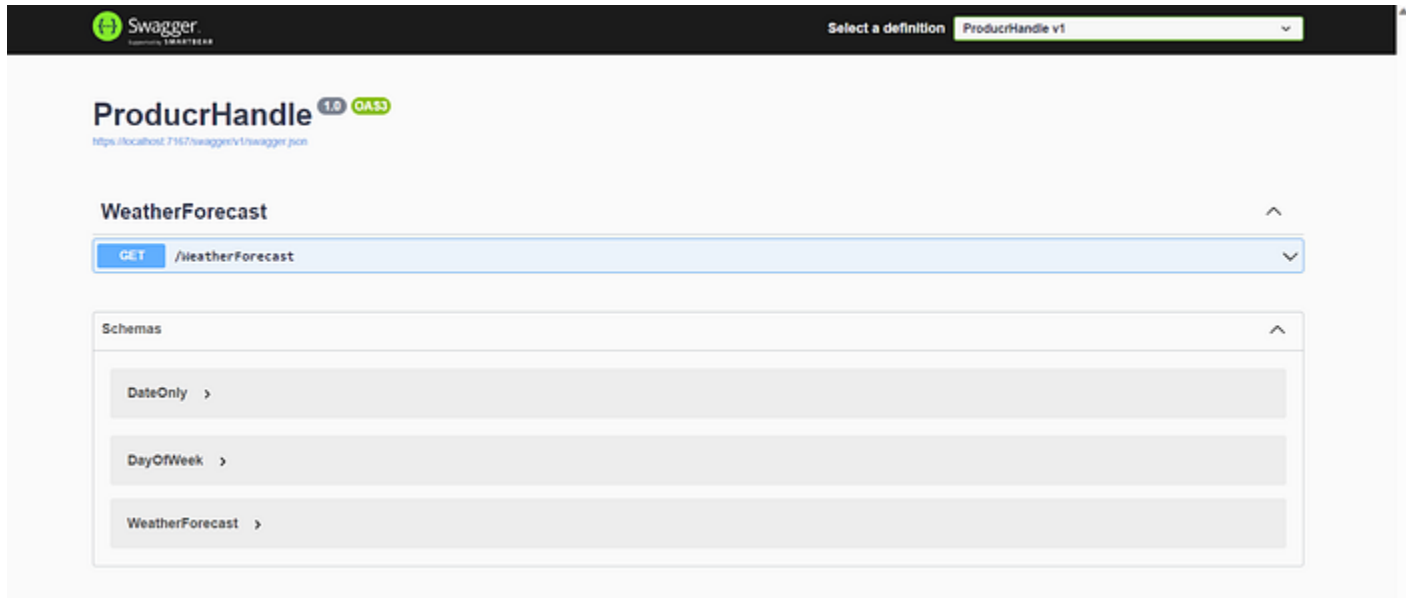


Assign a name and location to the project for saving.



Retain the automatically provided additional information.

Once the .NET project is created, run the solution to ensure it works without errors. The output should resemble the following:



### Create a New Database with two tables

Next, proceed to create a database named “Stock” containing two tables, “Products” and “AdminInfo.” You can utilize the following SQL script to generate the necessary tables:

```
CREATE TABLE Products (  
    ProductId INT IDENTITY(1,1) PRIMARY KEY,  
    ProductName VARCHAR(100) NOT NULL,  
    UnitPrice DECIMAL NOT NULL,  
    StockQuantity INT NOT NULL,  
    SupplierName VARCHAR(50),  
    Weight DECIMAL,  
);
```

```
Create Table AdminInfo(  
    AdminId Int Identity(1,1) Not null Primary Key,  
    FirstName Varchar(30) Not null,  
    LastName Varchar(30) Not null,  
    UserName Varchar(30) Not null,  
    Email Varchar(50) Not null,  
    Password Varchar(20) Not null  
);
```

```
INSERT INTO Products (ProductName, UnitPrice, StockQuantity, SupplierName, Weight)  
VALUES
```

```
('Product1', 29.99, 100, 'SupplierA', 1.5),  
( 'Product2', 19.99, 150, 'SupplierB', 2.0),  
( 'Product3', 39.99, 80, 'SupplierC', 1.2),  
( 'Product4', 49.99, 120, 'SupplierA', 1.8),  
( 'Product5', 14.99, 200, 'SupplierB', 1.0);
```

```
INSERT INTO AdminInfo (FirstName, LastName, UserName, Email, Password)  
VALUES
```

```
('John', 'Doe', 'john_doe', 'john@example.com', '$admin@2024');
```

## Install NuGet Packages

Add the following NuGet packages to the solution:

**Microsoft.EntityFrameworkCore.SqlServer** — Enables Entity Framework Core to work with SQL Server databases, providing seamless integration and interaction with the database.

**Microsoft.VisualStudio.Web.CodeGeneration.Design** — Includes design-time components for code generation, aiding in the scaffolding of controllers and views within the ASP.NET Core project.

**Microsoft.EntityFrameworkCore.Tools** — Provides additional tools for Entity Framework Core, facilitating tasks such as migrations and database updates from the command line.

**Microsoft.EntityFrameworkCore** — The core Entity Framework package that allows developers to interact with databases using .NET objects, simplifying data access and manipulation.

**System.IdentityModel.Tokens.Jwt** — Adds support for JSON Web Tokens (JWT), which is essential for implementing secure authentication and authorization in your ASP.NET Core Web API.

**Microsoft.AspNetCore.Authentication.JwtBearer** — Integrates JWT-based authentication into your ASP.NET Core application, allowing secure validation and authentication of users based on JWTs.

## Create model classes automatically using the database

First of all, Make sure `<InvariantGlobalization>` is false before connecting our database with the project.

```
<PropertyGroup>  
  <TargetFramework>net8.0</TargetFramework>  
  <Nullable>enable</Nullable>  
  <ImplicitUsings>enable</ImplicitUsings>  
  <InvariantGlobalization>False</InvariantGlobalization>  
</PropertyGroup>
```

```
</PropertyGroup>
```

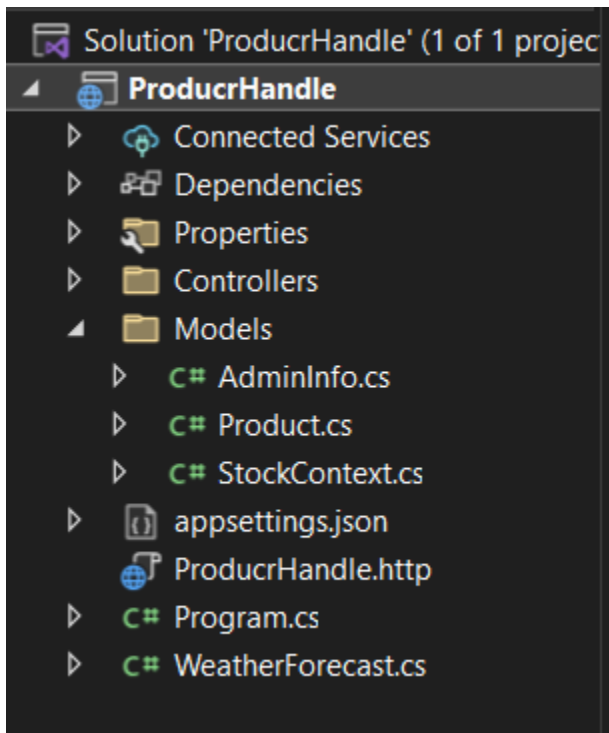
Go to,

Tools → NuGet Package Manager → Package Manager Consoler

Now run this code on the Package Manager Console

```
Scaffold-DbContext "Server=server_name;Database=your_db_name;Trusted_Connection=True;
Encrypt=False" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

Subsequently, it will automatically generate the classes AdminInfo.cs, Product.cs, and StockContext.cs within the 'Models' folder.



Add this line into the program.cs file to configure your database context.

```
builder.Services.AddDbContext<yourDbContextname>(  
    o => o.UseSqlServer(builder.Configuration.GetConnectionString("SqlServer")));
```

```
builder.Services.AddEndpointsApiExplorer();  
builder.Services.AddSwaggerGen();  
  
builder.Services.AddDbContext<StockContext>(  
    o => o.UseSqlServer(builder.Configuration.GetConnectionString("SqlServer")));  
  
var app = builder.Build();
```

Then add the connection string inside the appsettings.json file

```
"ConnectionStrings": {
```

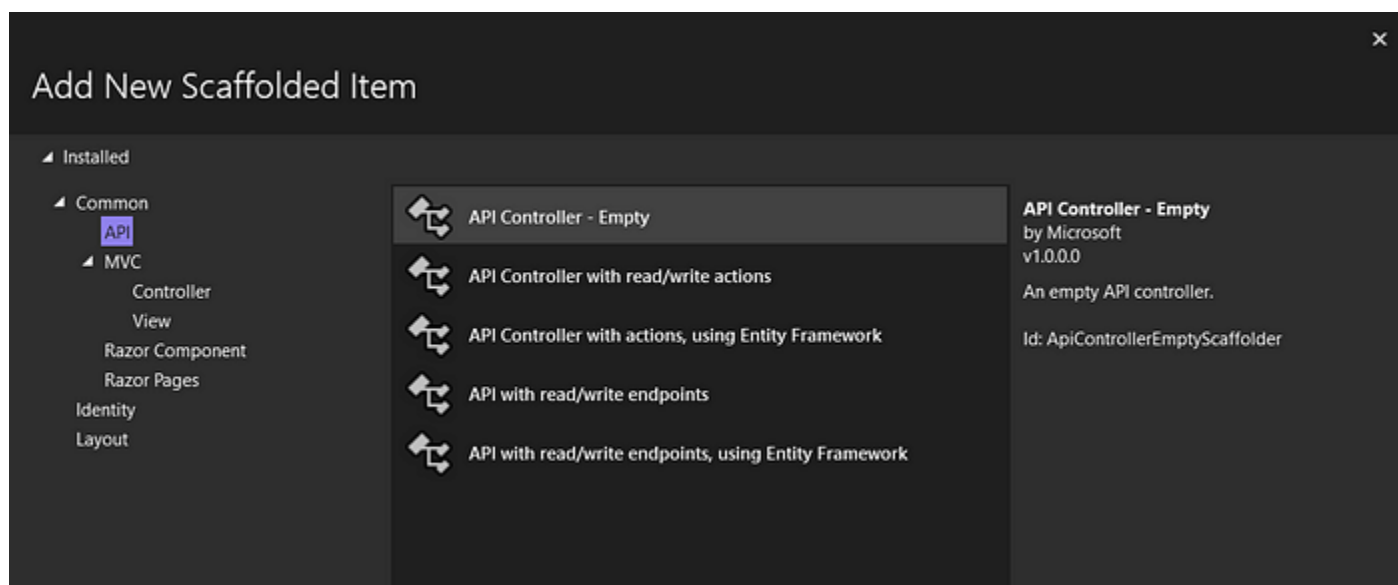
```
"SqlServer": "Data Source=*****;Initial Catalog=Stock;Integrated
Security=True;Encrypt=True;TrustServerCertificate=True"
}
```

## Create the controller class

Now right-click the controller folder then,

Add → controller

Then select **Common** → **API** → **API controller**



The controller class should look like this,

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace ProducrHandle.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    0 references
    public class ProductController : ControllerBase
    {
    }
}
```

Add a constructor to the class

```
private readonly StockContext _context;
```

```
public ProductController(StockContext context)
{
    _context = context;
}
```

These methods will serve as controllers for each CRUD operation.

Get all the data from the table

```
[HttpGet]
public async Task<IEnumerable<Product>> Get()
{
    return await _context.Products.ToListAsync();
}
```

Get data by the given id

```
[HttpGet("id")]
[ProducesResponseType(typeof(Product), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<ActionResult> GetById(int id)
{
    var product = await _context.Products.FindAsync(id);
    return product == null ? NotFound() : Ok(product);
}
```

Update data by the given id

```
[HttpPut("{id}")]
public async Task<ActionResult> Update(int id, Product product)
{
    if (id != product.ProductId) return BadRequest();
    _context.Entry(product).State = EntityState.Modified;
    await _context.SaveChangesAsync();
    return NoContent();
}
```

Add new data to the table



```
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
public async Task<IActionResult> Create(Product product)
{
    await _context.Products.AddAsync(product);
    await _context.SaveChangesAsync();
    return CreatedAtAction(nameof(GetById), new { id = product.ProductId }, product);
}
```

Delete data by the given id

```
[HttpDelete("{id}")]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<IActionResult> Delete(int id)
{
    var productToDelete = await _context.Products.FindAsync(id);
    if (productToDelete == null) return NotFound();
    _context.Products.Remove(productToDelete);
    await _context.SaveChangesAsync();

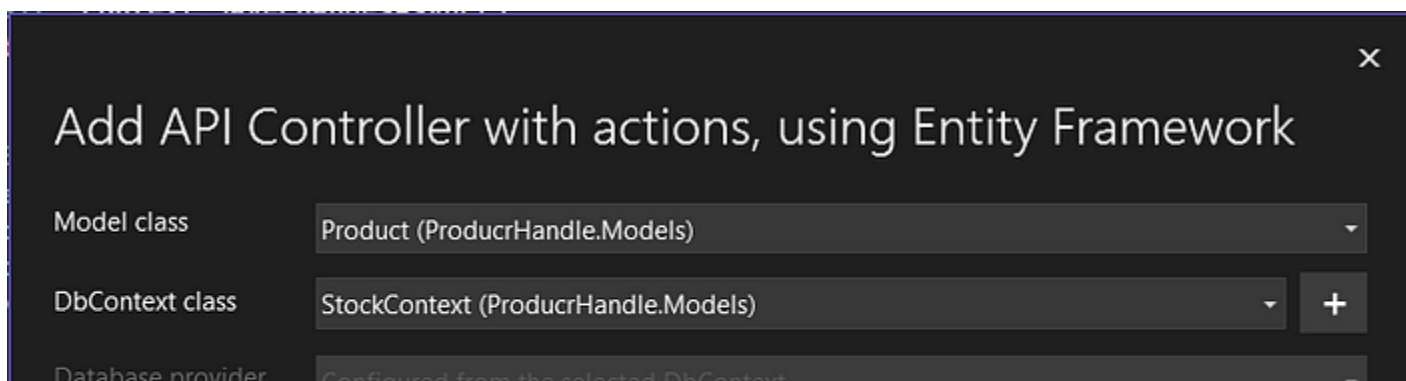
    return NoContent();
}
```

*Creating the controller class can be done automatically.*

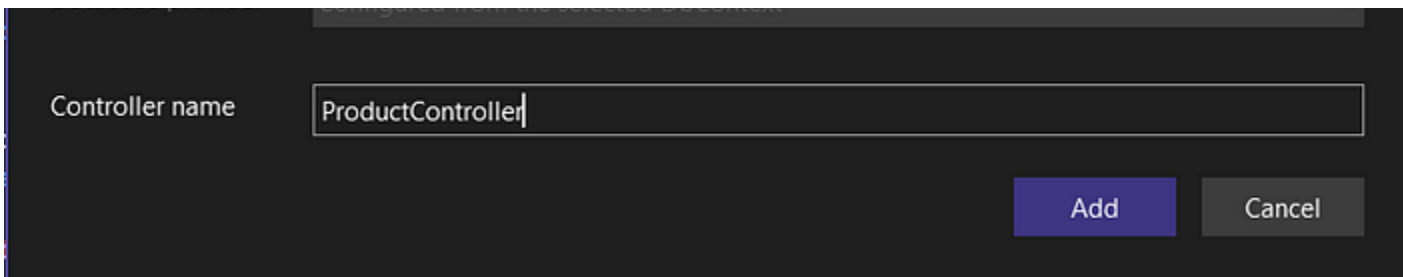
*Right-click the Controllers folder, choose, Add, and then click Controller.*

*Select API Controller with actions using the Entity Framework template.*

*Choose the Products model class and InventoryContext context class, and then name the control ProductsController.*







Controller name

Add Cancel

*The following APIs will be created:*

*To list all products: HTTP Get method*

*Get product detail: HTTP Get method*

*Update product detail: HTTP Put method*

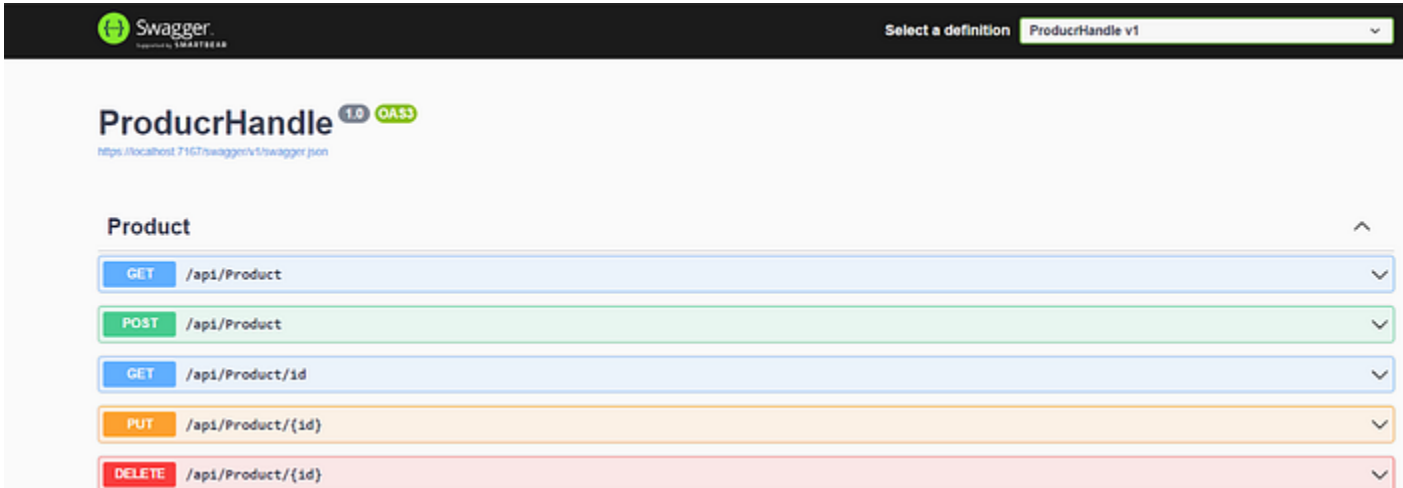
*Create product: HTTP Post method*

*Delete product: HTTP Delete method*

*And we can modify these methods as we want.*

## Test using Swagger & postman

When we run the solution, we will receive an output like this.

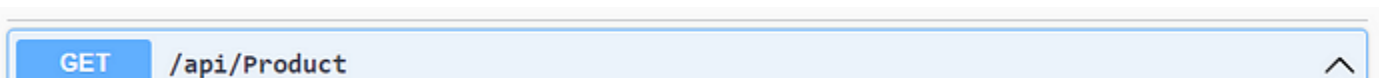


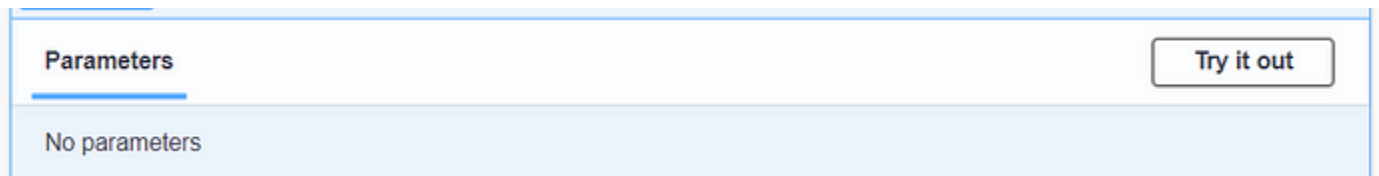
We can verify the functionality of our methods by clicking on each one, or alternatively, we can use Postman to test our solution.

## Test the method that lists all the Products,

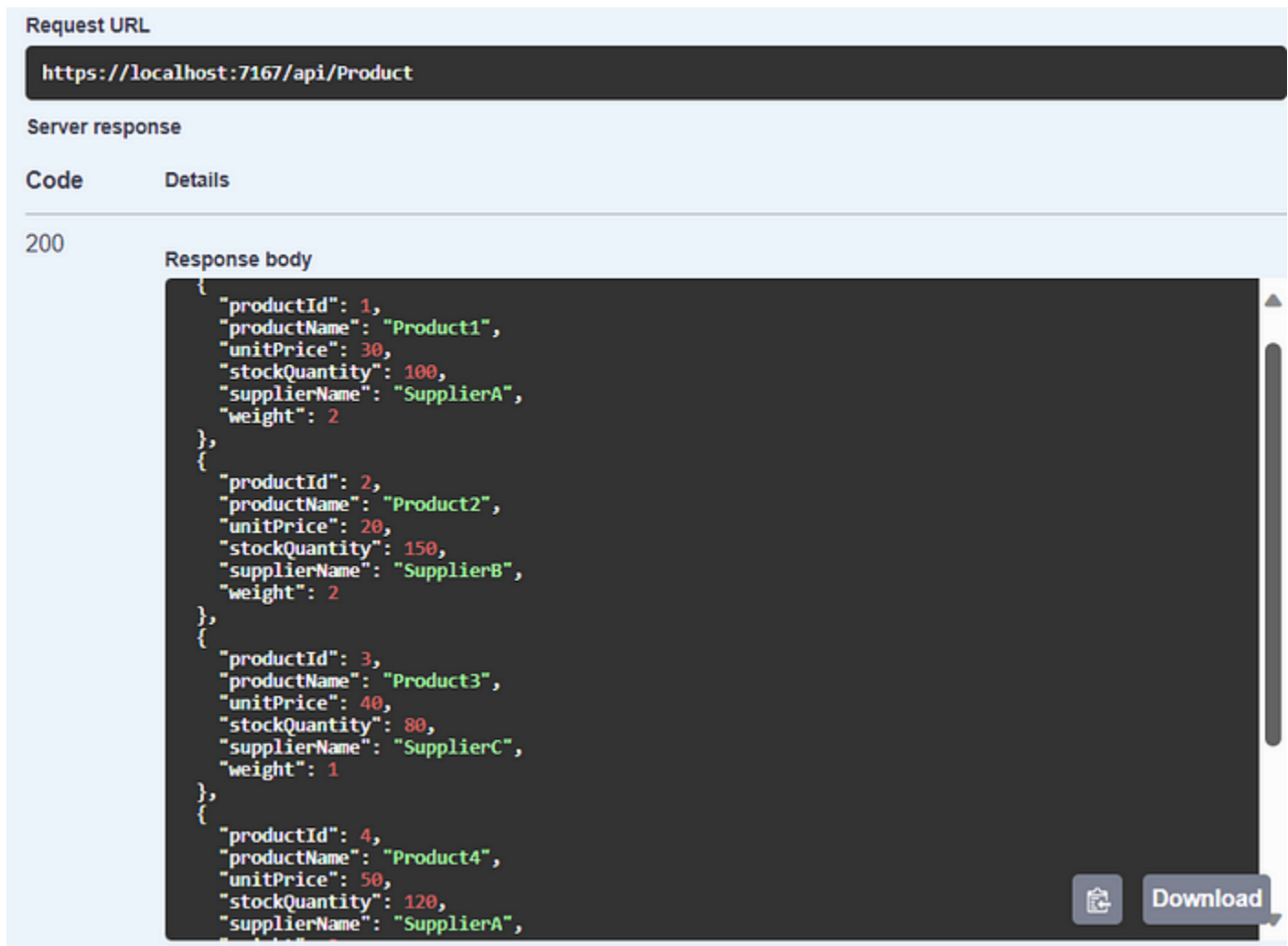
### Using Swagger:

Click on the first 'GET' and select 'Try it out.'





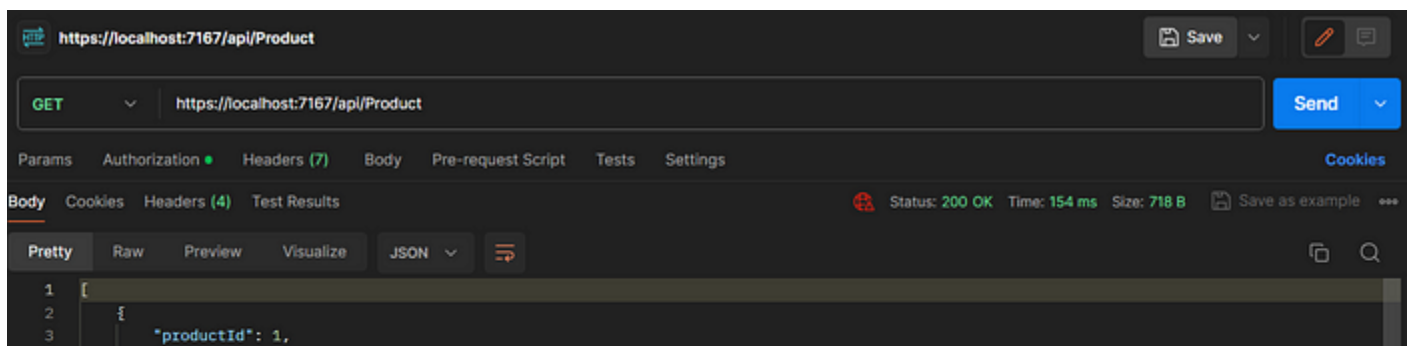
Subsequently, click the 'Execute' button to retrieve the list of products stored in the database.



## Using Postman:

Open Postman and enter the following endpoint: <https://localhost:7167/api/Product>

Choose the GET method and click 'Send.' Now, all the products will be listed, as shown in the following screenshot.



```
4      "productName": "Product1",
5      "unitPrice": 30,
6      "stockQuantity": 100,
7      "supplierName": "SupplierA",
8      "weight": 2
9    },
10   {
11     "productId": 2,
12     "productName": "Product2",
13     "unitPrice": 20,
14     "stockQuantity": 150,
15     "supplierName": "SupplierB",
16     "weight": 2
17   },
18   {
19     "productId": 3,
20     "productName": "Product3",
21     "unitPrice": 40,
22     "stockQuantity": 80,
23     "supplierName": "SupplierC",
24     "weight": 1
25   },
26   {
27     "productId": 4,
```

**Test the method that lists the Product details by the given ID,**

### Using Swagger:

Select the 'GET' method that requires an ID. Click 'Try it out,' then enter the desired product ID for details and click the 'Execute' button.

GET /api/Product/id

Parameters

Cancel

| Name | Description |
|------|-------------|
| id   |             |

integer(\$int32)  
(query)

3

Execute

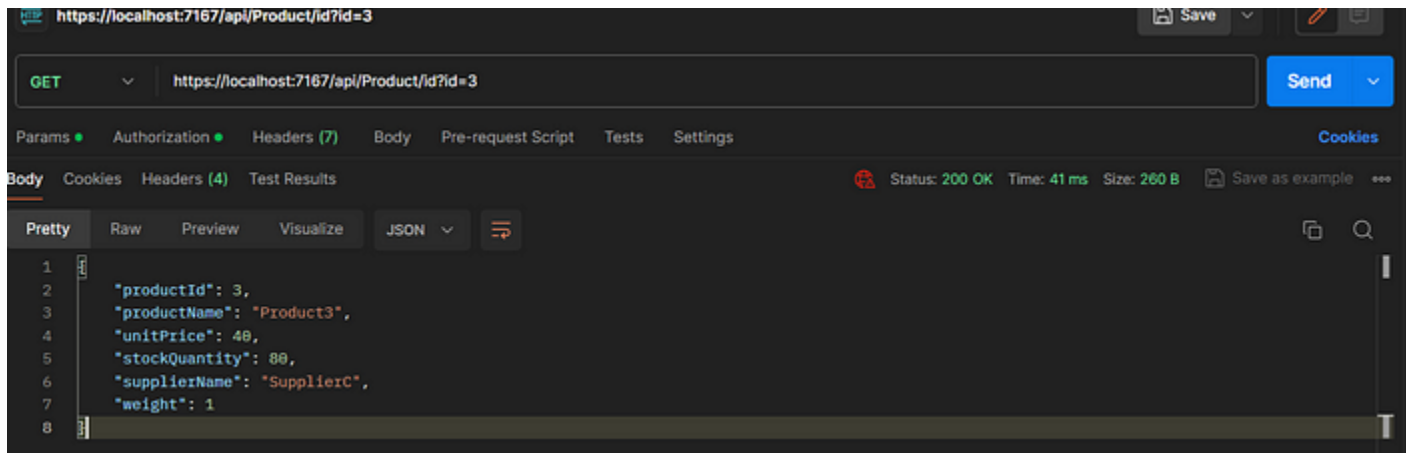
It will provide the details of the product id=3 as the response.

| Code | Details   |
|------|---|
| 200  | <p>Response body</p> <pre>{   "productId": 3,   "productName": "Product3",   "unitPrice": 40,   "stockQuantity": 80,   "supplierName": "SupplierC",   "weight": 1 }</pre> <p>Download</p> |

## Using Postman:

Open Postman and enter the following endpoint: <https://localhost:7167/api/Product/id?id=3>

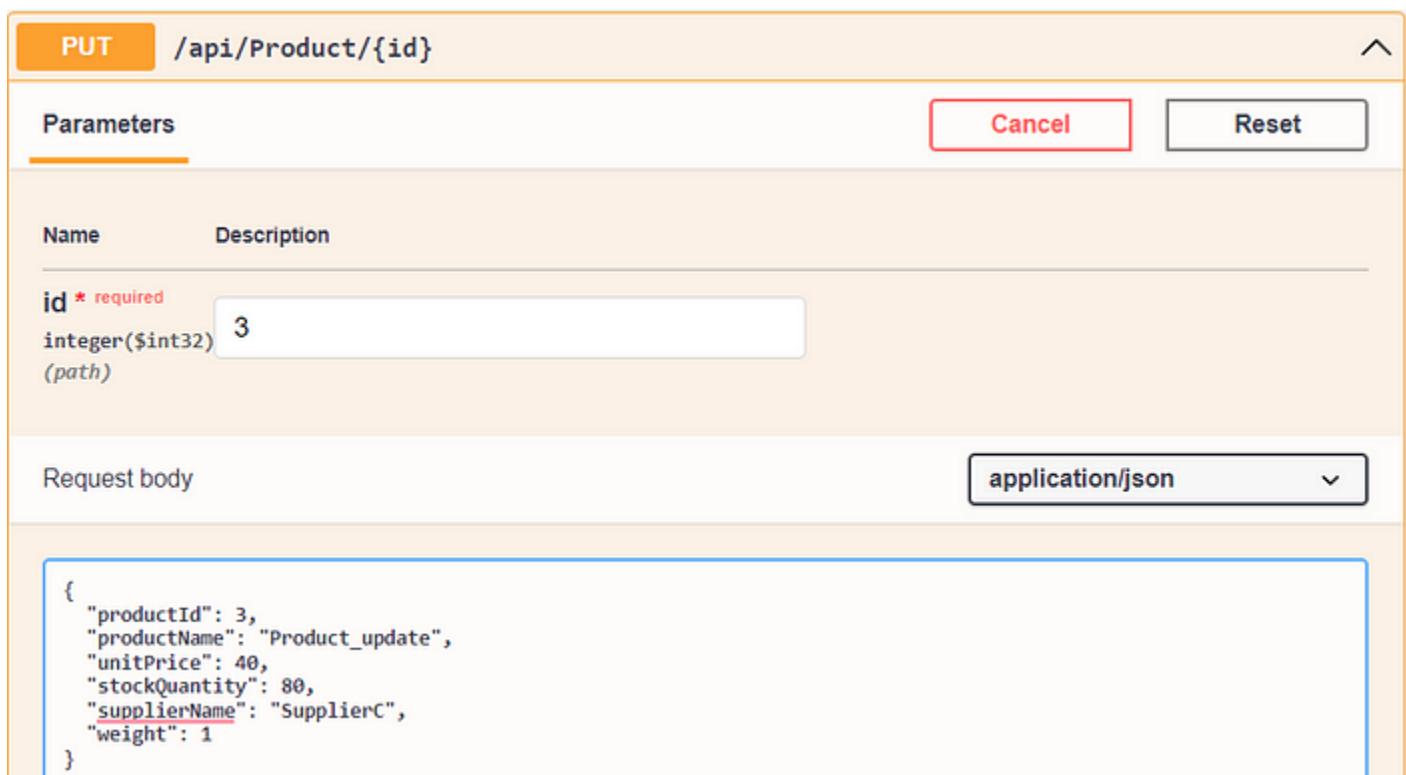
Select the GET method and click 'Send.' Now, you can view the details of the product. In this example, the product ID is 3.



## Test the method that updates the Product details by the given ID,

### Using Swagger:

First, retrieve the details of the product you need to update using the GET by ID method. Copy the details into the request body for the 'PUT' operation and make the desired modifications. For instance, I changed the productName from 'Product3' to 'Product\_update,' then added the ID in the top request field and clicked the 'Execute' button.



The data will be updated in the database.

## Using Postman:

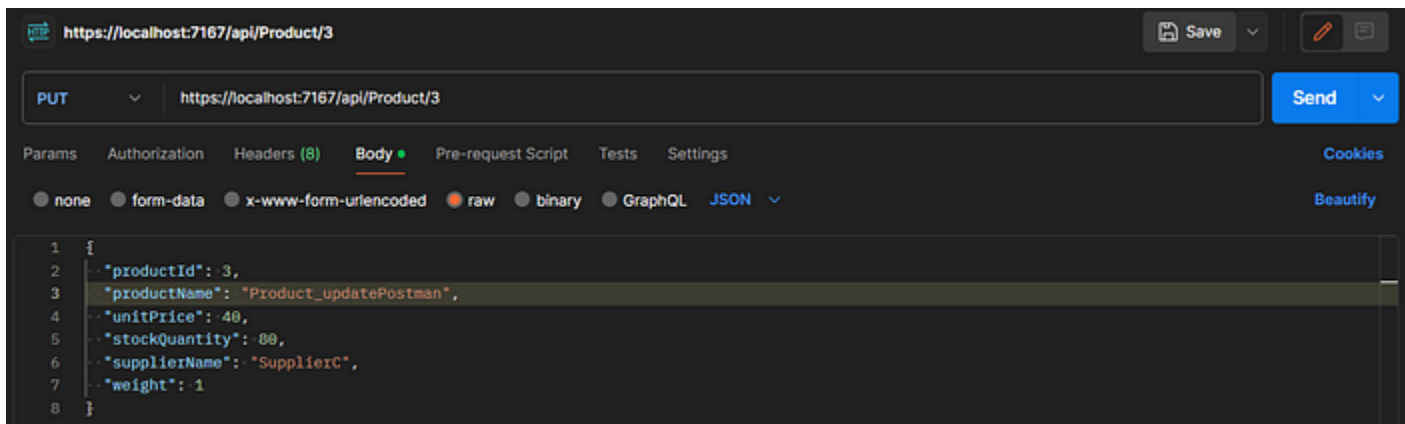
Similar to Swagger, first, retrieve the details of the product you want to update.

In the 'Body' section, select 'Raw,' choose the JSON (application/json) type, and paste the product details.

Make the necessary changes.

Enter this endpoint: <https://localhost:7167/api/Product/3>

Choose the 'PUT' method and click 'Send.' The data will be saved in the database.

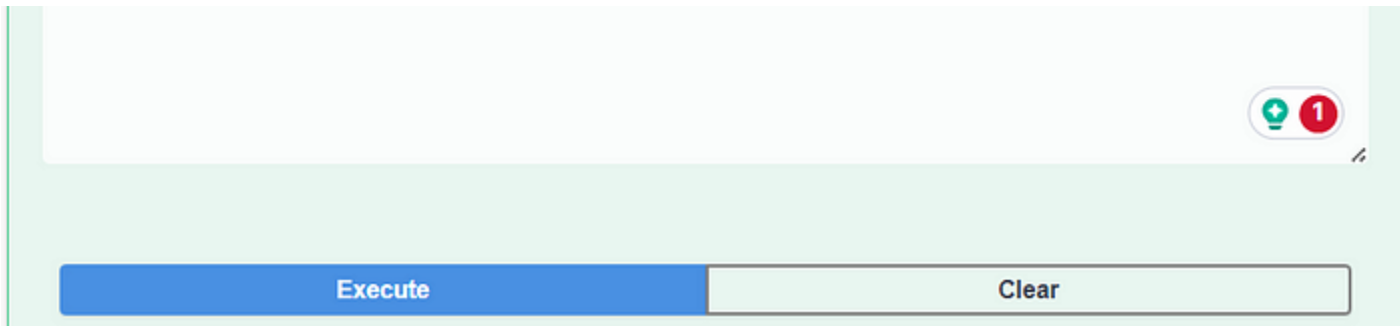


## Test the method that will create a new product,

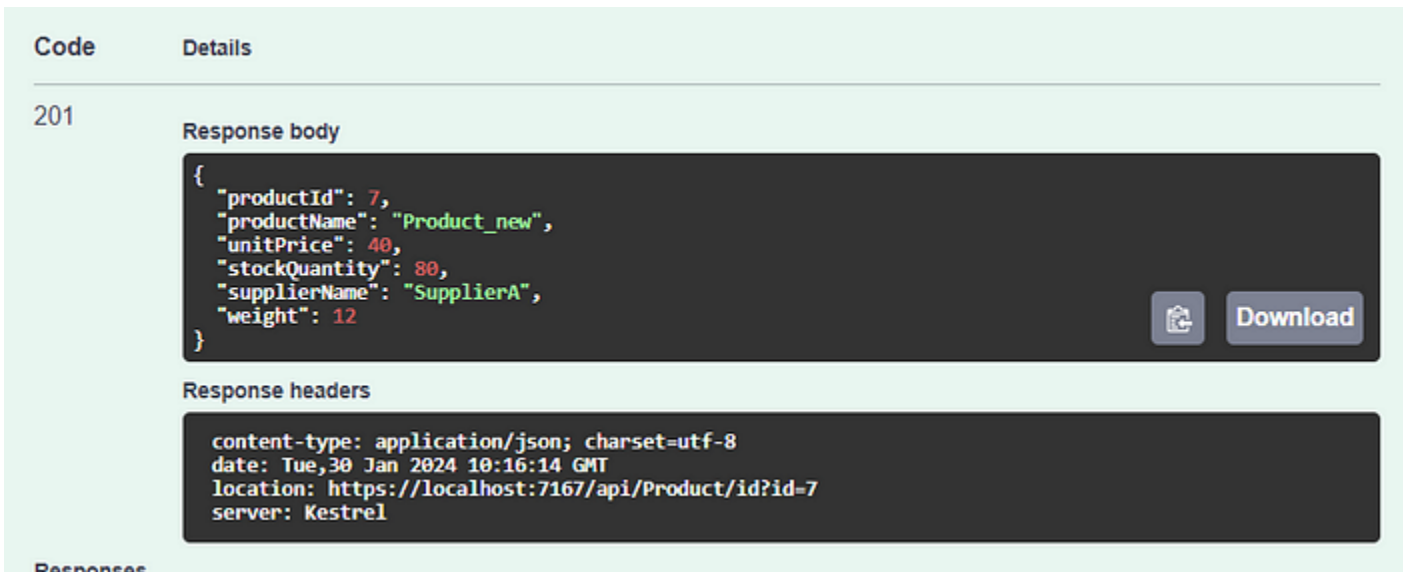
### Using Swagger:

Add the new product details inside the request body and click the 'Execute' button.





The new product will be added to the database.



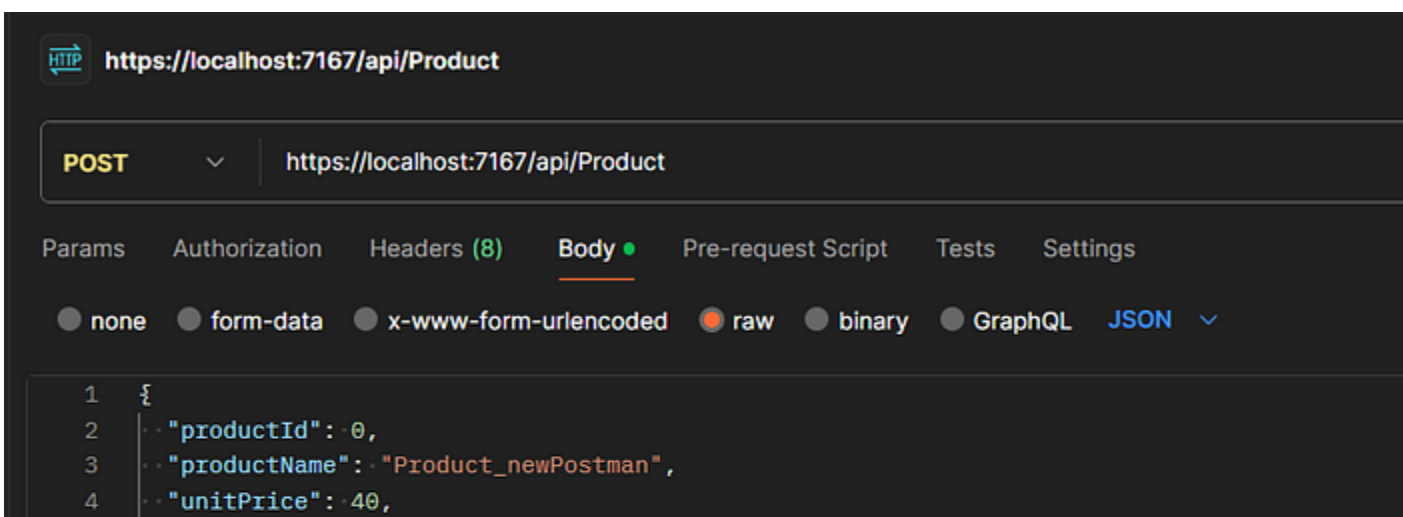
## Using Postman:

First, retrieve the structure of the product table.

In the 'Body' section, select 'Raw,' choose JSON (application/json) as the type, paste the structure, and make the necessary changes.

Enter this endpoint: <https://localhost:7167/api/Product>

Choose the 'POST' method and click 'Send.' The data will be saved in the database



```
5  {
6    "stockQuantity": 80,
7    "supplierName": "SupplierA",
8    "weight": 10
9  }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1  {
2    "productId": 8,
3    "productName": "Product_newPostman",
4    "unitPrice": 40,
5    "stockQuantity": 80,
6    "supplierName": "SupplierA",
7    "weight": 10
8  }
```

**Test the method that deletes the Product details by the given ID,**

### Using Swagger:

Select 'Delete' and provide the ID of the product that you want to delete, then click the 'Execute' button. The data will be deleted from the database.

**DELETE** /api/Product/{id}

Parameters Cancel

| Name             | Description |
|------------------|-------------|
| id * required    |             |
| integer(\$int32) | 2           |
| (path)           |             |

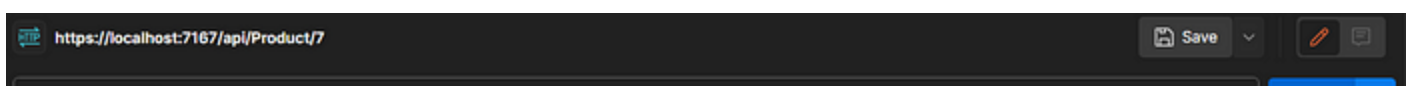
Execute Clear

### Using Postman:

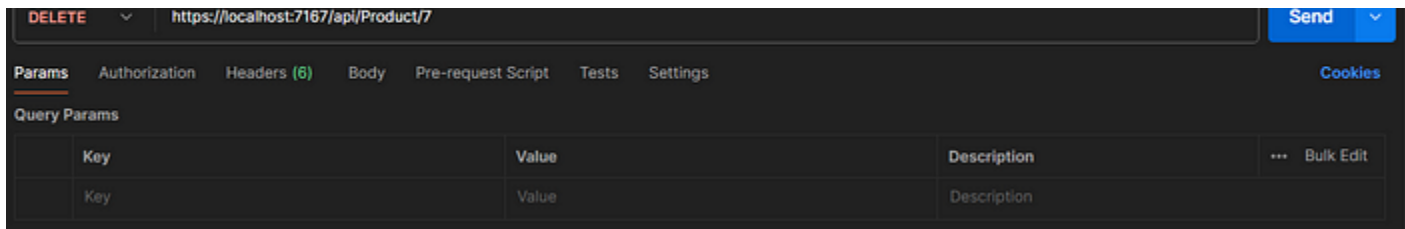
Enter this endpoint in Postman: <https://localhost:7167/api/Product/7>

Choose the 'DELETE' method and click 'Send.'

Now, the product will be deleted from the database.







## Conclusion:

Congratulations on successfully creating a .NET Core Web API with Entity Framework! In this guide, we've walked through the essential steps, from setting up the project using Visual Studio 2022 to creating a database with two tables, and automatically generating model classes. By adding controllers and implementing CRUD operations, you've laid the foundation for a powerful API.

Remember, this guide is just the beginning. As you explore and modify the provided methods, you'll further enhance your skills in crafting robust APIs tailored to your specific needs. Utilizing tools like Swagger and Postman for testing ensures the reliability and effectiveness of your API.

So, whether you're building the next big thing or just honing your coding skills, keep diving into the world of .NET Core. Happy coding!