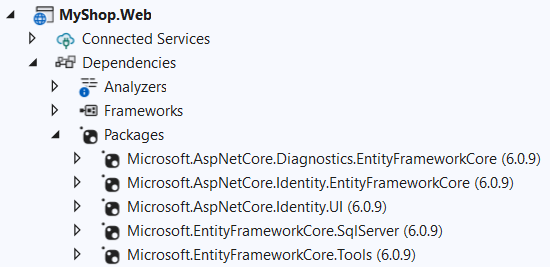
***MyShop, a state-of-the-art webshop, Part 3***

In this *3rd MyShop exercise* we'll start from the final version of the exercise of *Part 2*. In this version of our webshop we provide an API for the orders and products.

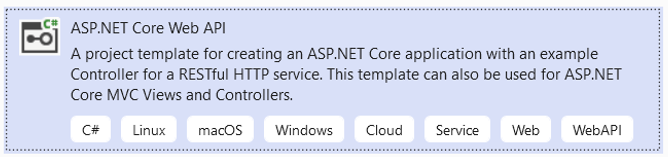
***1. Preparation***

As with regular ASP.NET MVC Controllers, API Controllers can also be generated (scaffolding). To be able to make use of this scaffolding we need to work with the correct EF Core versions.

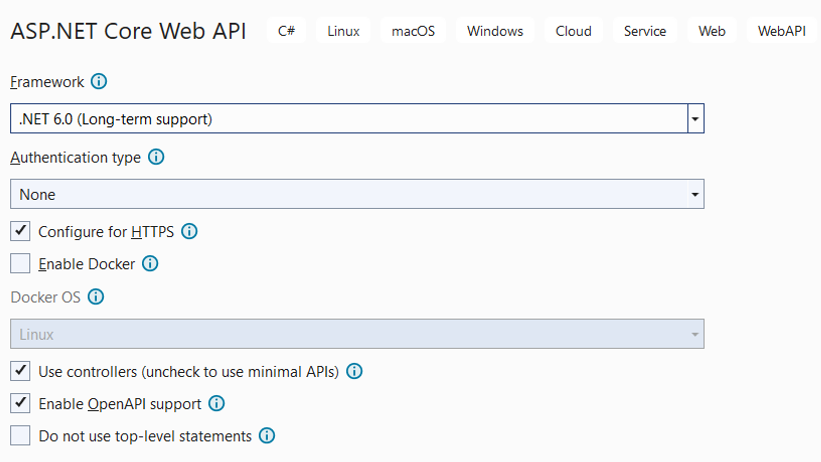
Check the *Packages* of the **MyShop.Web** project and make sure they match the following screenshot (probably version 6.0.10):



Now add a new **Web API** Project to the **MyShop** solution:



Name: **MyShop.API**

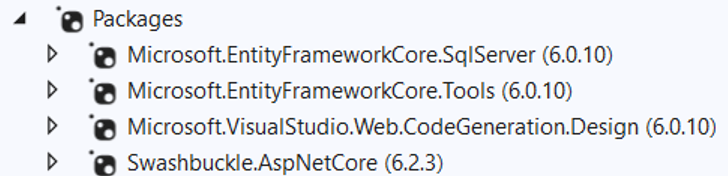


Add ‘MyShop.Domain’ and ‘MyShop.Infrastructure’ as project reference to this API project. We need those libraries because we will make use of the Repository and UoW patterns.

***2. Configure MyShop.API to enable data access through repo/uow***

The API must have access to the database. Luckily we can reuse all the logic that we’ve written (repo, uow) from the MyShop.Infrastructure. Complete the followings steps:

* Copy the connection string from the one in **MyShop.Web** and put it in the *appsettings.json* file from the **MyShop.API** 🡪 this connection string is needed for connecting to the database
* Add the following NuGet packages (make sure the versions are the same as in the webproject!):



* Add the ShoppingContext and the IUnitOfWork interface to the Dependency Injection container like in the **MyShop.Web** app (*Program.cs*)

***3. Create a controller for the Products (manually)***

Add a new **empty** API Controller named **ProductsController**. Inject the IUnitOfWork interface in the constructor. Create the following API methods:

|  |  |  |
| --- | --- | --- |
| **API** | **Name** | **Purpose** |
|  | GetProducts | Returns all the products from the database |
|  | GetProduct | Returns a product based on its id |
|  | PostProduct | Adds a new product and returns it |

Tip: return ActionResults instead of just the object. Returning the correct ActionResult will add the corresponding Http status code to the response. For the post:



This will return a **201 Created** response, containing the returned product which we get from the other action method ‘**GetProduct**’.

***4. Create a controller for the Orders (scaffolding)***

We also want to create endpoints for the orders from our shop. This time we won’t create the API controller ourselves but we will make use of the scaffolding tools.

* Generate an API controller with actions using Entity Framework

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Afbeelding met tekst

Automatisch gegenereerde beschrijving

* A generated API Controller works by default directly on the DbContext and gives us async methods.
* At this moment we don’t include any related table (customer, orderlines, products), but we do want to include this in our response!

We need some **refactoring**!

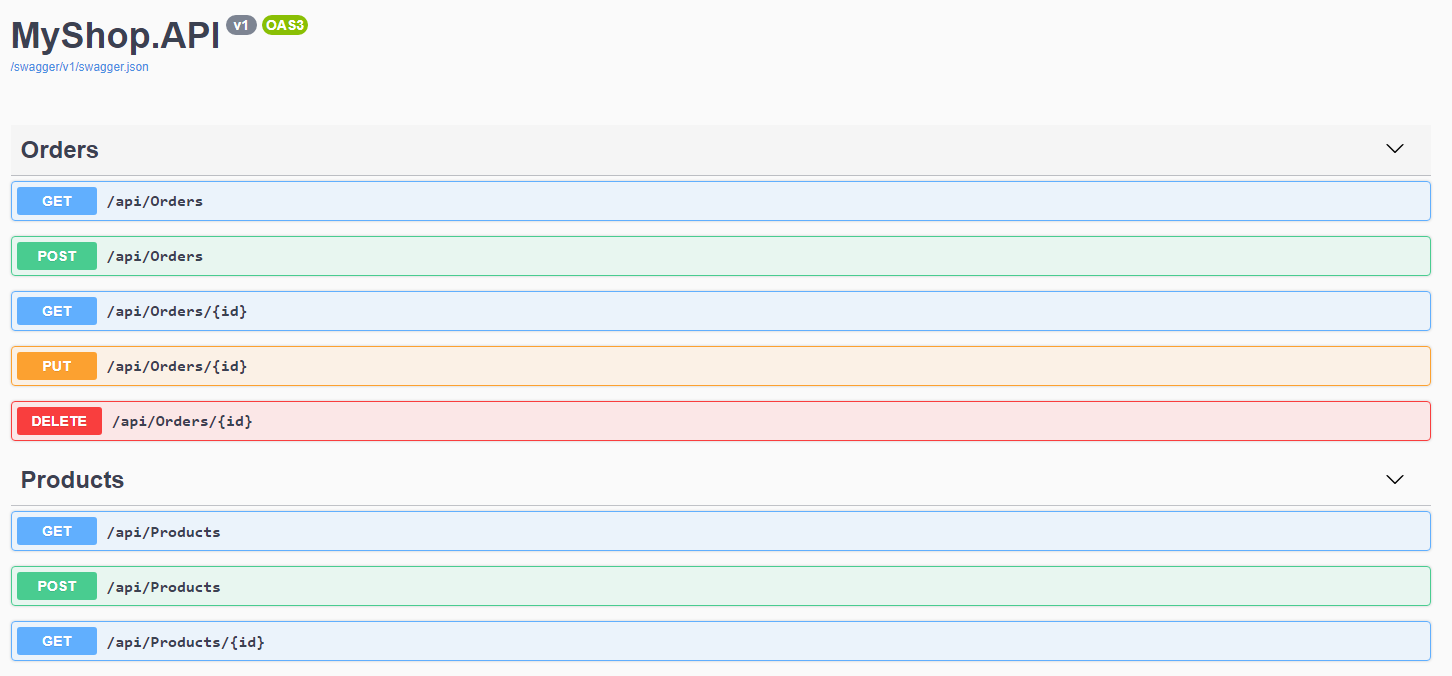
***5. Refactoring***

* Extend the GenericRepository (and interface) with async methods for all the methods where you query the database
* Extend the UnitOfWork (and interface) with an async function to save all the changes
* Override the AllAsync() method in the OrderRepository to make sure that we include the customer, orderlines and products.
* Add a delete function to the IRepository and implement it in the GenericRepository
* Get rid of the ShoppingContext in the OrdersController and inject the IUnitOfWork interface
* Refactor the code so the UnitOfWork is used and that the async methods are called

Tip: returning objects with relationships often results in the famous ‘*A possible object cycle was detected*’ error. Make sure to fix this the way we’ve seen in class!

***6. Expected result***

All methods (Swagger documentation)



Getting all orders (with included tables)

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Afbeelding met tekst

Automatisch gegenereerde beschrijving

***7. Consume the web api***

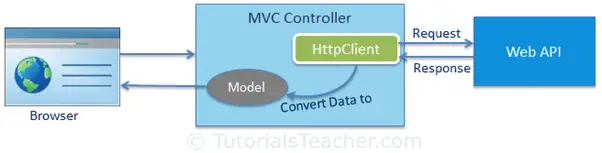
In the previous section, we created a Web API that handles HTTP GET, POST, PUT and DELETE requests. Here, we will see how to consume (access) Web API for CRUD operation. We focus in this exercise on retrieving the list of all orders.

Web API can be accessed in the server side code in .NET and also on client side using JavaScript frameworks such as jQuery, AnguarJS, KnockoutJS and Vue.js. Access Web API in a razor page using Vue.js is the topic of the extra exercise in part 8.

Here, we will consume our Web API in the controllers from the ASP.NET MVC Shop.Web application.

To consume Web API in ASP.NET MVC server side we use **HttpClient** in the MVC controller. **HttpClient** sends a request to the Web API and receives a response. We then need to convert (deserialize) response data that came from Web API to a model and finally render it into a view.

The following figure illustrates consuming Web API at Server side ASP.NET MVC



Goal is to change the Index method in de OrderController from the Shop.Web application. Instead of accessing the UnitOfWork immediately from inside this controller to get an overview of all orders we now make an API call to https://localhost: xxxx/api/orders.

The code you need looks like this:

public async Task<IActionResult> IndexAsync()

{

IEnumerable<Order> orders = null;

using (var client = new HttpClient())

{

client.BaseAddress = new Uri("https://localhost:7045");

HttpResponseMessage response = await client.GetAsync("api/Orders");

if (response.IsSuccessStatusCode)

{

var ordersResponse = response.Content.ReadAsStringAsync().Result;

//deserialize the response from webapi and storing into orders IEnumerable

orders = JsonConvert.DeserializeObject<IEnumerable<Order>>(ordersResponse);

}

return View(orders);

}  
}

In order to test your final result, you must make sure that when you run your solution the shop.api project and the shop.web project both start. Click on the solution properties and mark them both as startup projects.

A screenshot of a computer

Description automatically generated with medium confidence

When you now click on the orders link in the web project, the view isn’t changed but the data are passed via the web api.

A picture containing text, screenshot, font, line

Description automatically generated***8. Extra***

Extend your Vue.js knowledge and try to build a new page in the MyShop.Web project where you get all the orders from calling the API you just created but this time with Vue.js! Calling the API with Vue.js is done in the View, not in the Controller!

Tip 1: calling your API from another url will be blocked by default. Configure CORS to fix this!

Tip 2: your JavaScript code might look like this:

Afbeelding met tekst

Automatisch gegenereerde beschrijving