Exercise

OdeToFood – Part 1: REST Api

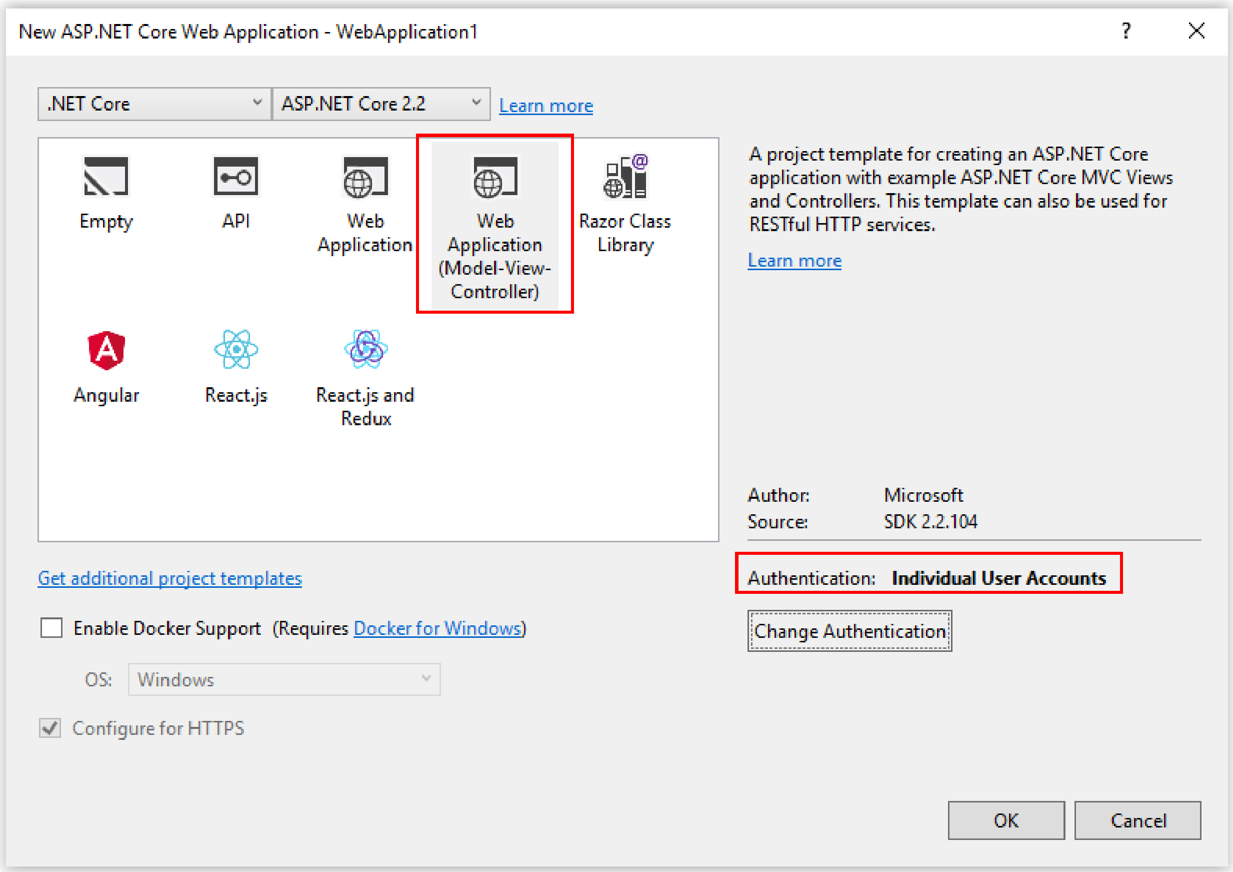
May 2019

We're going to build a web application that has some (html) pages that enable visitors to show, create and edit restaurants and reviews about that restaurants.

The application will also have a REST api for other applications to link to (e.g. a native Android app). In the first part we will focus on the api.

Setup

Create a new ASP.NET Core Webapplication project called "OdeToFood". Use the "Web Application (Model-View-Controller)" template.
Choose for "Individual user accounts" (store user accounts in-app).


New ASP.NET Core Web Application - WebApplication1

.NET Core · ASP.NET Core 2.2

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

Author: Microsoft
Source: SDK 2.2.104

Authentication: **Individual User Accounts**

Enable Docker Support (Requires Docker for Windows)

OS: Windows

Configure for HTTPS

Part 1: REST Api

The OdeToFood Api offers a REST interface to

- Create, read, update and delete restaurants in an OdeToFood databse
- To create, read, update and delete restaurant reviews

The API should be RESTful. The table below shows how the API should respond to certain http requests:

Http Verb	CRUD	Expected response
POST	Create	201 (Created), 'Location' header with link to created resource Created resource in the body
GET	Read	200 (OK) Requested resource (list or single item) in the body
PUT	Update / Replace	200 (OK) and empty body 404 (Not Found) if resource could not be found
DELETE	Delete	200 (OK) and empty body 404 (Not Found) if resource could not be found

Step 1 – create the Restaurants API controller

1. Create a Folder “Api” in the “Controllers” folder.
2. In this folder, add an “RestaurantsController” (Api Controller - Empty).
3. Add a Class Library project “OdeToFood.Data” to the solution. This project will be our data layer.
    - Add a folder “DomainClasses” to the data layer and add the following domain class in this folder:

C#

```
using System;
using System.ComponentModel.DataAnnotations;

namespace OdeToFood.Data.DomainClasses
{
    public class Restaurant
    {
        public int Id { get; set; }

        public string Name { get; set; }

        public string City { get; set; }

        public string Country { get; set; }
    }
}
```

4. Include a test project “OdeToFood.Tests” in the solution.
5. Create a folder “Controllers” in the test project, and a subfolder “Api”.
6. Add a “RestaurantsControllerTests” class in the “Controllers/api” folder of the test project.
7. Add tests (one by one) and implement the controller as you go (Red-Green-Refactor). You can use the domain classes as view models (normally you should use viewmodels to avoid overposting etc.):
    - Get_ReturnsAllRestaurantsFromRepository()
    - Get_ReturnsRestaurantIfItExists()
    - Get_ReturnsNotFoundIfItDoesNotExists()
    - Post_ValidRestaurantIsSavedInRepository()
    - Post_InValidRestaurantCausesBadRequest()
    - Put_ExistingRestaurantIsSavedInRepository()
    - Put_NonExistingRestaurantReturnsNotFound()
    - Put_InValidRestaurantModelStateCausesBadRequest()
    - Put_MismatchBetweenUrlIdAndRestaurantIdCausesBadRequest()
    - Delete_ExistingRestaurantIsDeletedFromRepository()
    - Delete_NonExistingRestaurantReturnsNotFound()
8. Tips
    - Use a mock for retrieving the restaurants from a repository (IRestaurantRepository).

- Use “IActionResult” as return type of the controller actions.
- Use a “[Setup]” method to create a new instance of the controller before each test.
- You should know how Web API validates action parameters to be able to implement some of the tests: https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-2.2

9. Add a “RestaurantDbRepository” that implements “IRestaurantRepository” and uses Entity Framework to retrieve / manipulate restaurants

    - Add an “OdeToFoodContext” class (derives from `IdentityDbContext`)

C#

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using OdeToFood.Data.DomainClasses;

namespace OdeToFood.Data
{
    public class OdeToFoodContext : IdentityDbContext<User, Role, int>
    {
        public DbSet<Review> Reviews { get; set; }

        public DbSet<Restaurant> Restaurants { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;
Database=OdeToFoodForApi; Integrated Security=True");
        }
    }
}
```

- Seed some restaurants by overriding ‘OnModelCreating’ and add a first migration using the package manager console:

```
PM>add-migrations initial
```

- Create the database using the package manager console:

```
PM>update-database
```

- Add a “RestaurantDbRepository” that implements “IRestaurantRepository”
    - Inject an instance of “OdeToFoodContext” into the constructor

- Use the context to retrieve / manipulate data

Note: You have to be careful when you implement the update method of the repository. The passed restaurant might not be tracked (attached) by the entity framework.

One solution is to find the original restaurant in de DB (by ID) and then copy the values from the passed restaurant to the original restaurant:

var original = _context.Restaurants.Find(restaurant.Id);

var entry = _context.Entry(original);

entry.CurrentValues.SetValues(restaurant);

10. Use Postman to compose http requests to the Api that create, read, update and delete restaurants.

Step 2 – Create the review API controller

Use the same methods as in step 1.

But now all action methods must be "async" in the review controller.

Domain class:

C#

```
using System.ComponentModel.DataAnnotations;

namespace OdeToFood.Data.DomainClasses
{
    public class Review
    {
        public int Id { get; set; }

        [Range(1, 10)]
        public int Rating { get; set; }

        public string Body { get; set; }

        public int RestaurantId { get; set; }
        public virtual Restaurant Restaurant { get; set; }

        [Required]
        public string ReviewerName { get; set; }
    }
}
```

This will be useful if you have a Review repository that is also asynchronous so that you can await database operations (that relatively take a long time to execute).

Make sure the Review repository implements the following interface:

C#

```
using OdeToFood.Data.DomainClasses;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace OdeToFood.Data
{
    public interface IReviewRepository
    {
        Task<IList<Review>> GetAllAsync();
        Task<Review> GetByIdAsync(int id);
        Task<Review> AddAsync(Review review);
        Task UpdateAsync(Review review);
        Task DeleteAsync(int id);
    }
}
```

Tips:

- Entity Framework offers asynchronous alternatives like
    - *SaveChangesAsync()*
    - ToListAsync()
    - FirstOrDefaultAsync()

Step 3

1. Add an Authentication Controller for the Api. You don’t have to use TDD or write tests to implement this controller (but you may if you want to).
2. Add a Register and a GetToken method.
3. Make sure each new user gets the role “User” in the “Register” action of the “AccountsController”.
4. Make sure the Restaurants and the Reviews Api Controllers are protected by a bearer token.

Step 4 (Optional) – Generic repositories

Challenge:

- Make the restaurant repository also asynchronous
- Remove the duplication in de repository classes by refactoring to one generic repository class that can be used for *Review* and *Restaurant*. (DRY)