How to Handle GET Requests

So far, we have created a repository pattern for interfacing with a database.

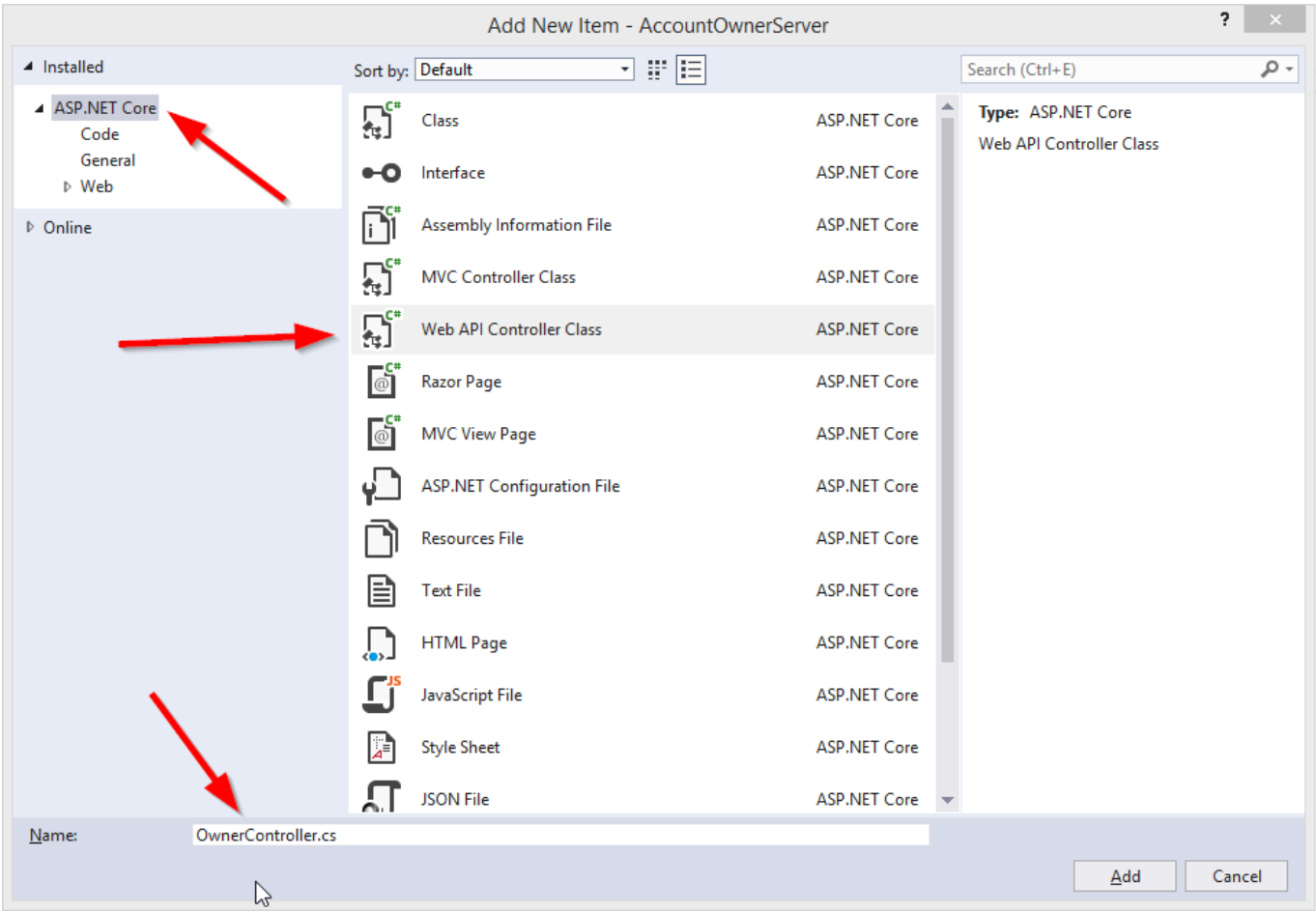
Now it's time to use that repository for business logic.

We are going to keep all the database logic inside the repository classes. Controllers will be responsible for handling requests, model validation and returning responses to the frontend part of the application.

By doing so, controllers won't be overwhelmed with database code, thus making the code easier to read and maintain.

Controllers and Routing in Web API

To create a controller, right-click on the `Controllers` folder in the main project and add the new item. Then from the menu choose Web API Controller Class and name it `OwnerController.cs`.



Delete all actions inside the controller class:

```
using Microsoft.AspNetCore.Mvc;

namespace AccountOwnerServer.Controllers
{
    [Route("api/[controller]")]
```

```
    [ApiController]
    public class OwnerController : ControllerBase
    {

    }
}
```

Every Web API controller class inherits from the `Controller` abstract class that provides all the necessary behavior for the derived class.

Also, above the controller class you can see this code:

```
[Route("api/[controller]")]
```

Web API routing routes incoming HTTP requests to a particular action method inside a Web API controller.

The type of routing we are using is called *attribute routing*.

Attribute routing uses attributes to map routes directly to action methods inside a controller. Usually, we place the base route above the controller class, as you can see in the `OwnerController` class. Similarly, for the specific action methods in the class, we create their routes right above them.

GetAllOwners GET Request in .NET Core

Our first action method will return all the owners from the database.

In the `IOwnerRepository` interface create a definition for the method `GetAllOwners`:

```
public interface IOwnerRepository
{
    IEnumerable<Owner> GetAllOwners();
}
```

Then implement that interface in the `OwnerRepository` class:

```
namespace Repository
{
    public class OwnerRepository: RepositoryBase<Owner>, IOwnerRepository
    {
        public OwnerRepository(RepositoryContext repositoryContext)
            :base(repositoryContext)
        {
        }

        public IEnumerable<Owner> GetAllOwners()
        {
            return FindAll()
```

```
                .OrderBy(ow => ow.Name);
        }
    }
}
```

Finally, you need to return all the owners by using the `GetAllOwners` method inside the Web API action.

The purpose of the action methods inside Web API controllers is not only to return the results. You need to pay attention to the status codes of your Web API responses as well. Additionally, you'll have to decorate your actions with HTTP verb attributes to map the type of HTTP request to that action.

Modify the `OwnerController`:

```
using Contracts;
using Microsoft.AspNetCore.Mvc;
using System;

namespace AccountOwnerServer.Controllers
{
    [Route("api/owner")]
    [ApiController]
    public class OwnerController : ControllerBase
    {
        private ILoggerManager _logger;
        private IRepositoryWrapper _repository;

        public OwnerController(ILoggerManager logger, IRepositoryWrapper
repository)
        {
            _logger = logger;
            _repository = repository;
        }

        [HttpGet]
        public IActionResult GetAllOwners()
        {
            try
            {
                var owners = _repository.Owner.GetAllOwners();

                _logger.LogInfo($"Returned all owners from database.");

                return Ok(owners);
            }
            catch (Exception ex)
            {
                _logger.LogError($"Something went wrong inside GetAllOwners
action: {ex.Message}");
                return StatusCode(500, "Internal server error");
            }
        }
```

```
        }
    }
```

First of all, we inject the logger and repository services inside the constructor. Then by decorating the `GetAllOwners` action with the `[HttpGet]` attribute, we are mapping this action to a GET request. Finally, we use both injected services to log messages and to get data from the repository class.

The `IActionResult` interface supports using a variety of methods that return not only a result but a status code as well. In this situation, the `OK` method returns all the owners and also the status code 200 which stands for OK. If an exception occurs, we will return an internal server error with the status code 500.

Because there is no route attribute right above the action, the route for the action `GetAllOwners` will be api/owner (`http://localhost:5000/api/owner`).

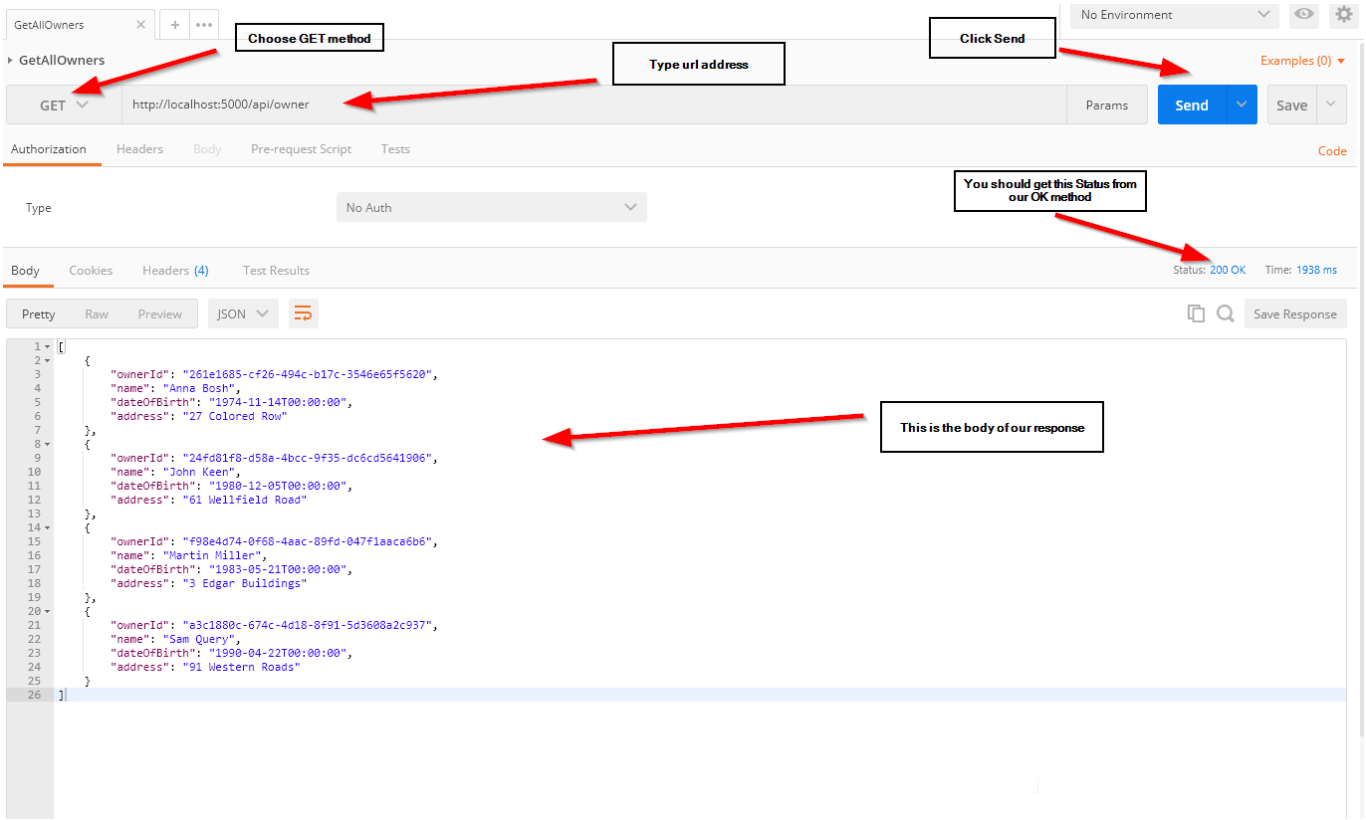
Code Permissions and Testing the Result

Right now, if you look at the repository structure, its classes inherit from the abstract `RepositoryBase<T>` class and also from its own interface, which then inherits from the `IRepositoryBase<T>` interface. With this hierarchy in place, by typing _repository.Owner. you are able to call custom methods from the `OwnerRepository` class and also all of the generic methods from the abstract `RepositoryBase<T>` class.

To avoid that type of behavior and to allow actions inside the controller to call only methods from the repository user classes, all you need to do is to remove `IRepositoryBase<T>` inheritance from `IOwnerRepository`. Consequently, only repository user classes will be able to call generic methods from the `RepositoryBase<T>` class. Likewise, action methods will only communicate only with repository user classes.

To check our results, we can use Postman to send requests to the application.

Start the application, start Postman and create a request:



If you look at the model classes, you'll notice that all properties have the same name as the database columns they are mapped to. But we can have properties with different names than the columns they map to. To achieve that, we need to use attribute `[Column]`.

Change the property names from `AccountId` and `OwnerId` to just `Id` in the `Owner` and `Account` classes. Also, add the `[Column]` property which will map the `Id` property to the right column in the database:

```
[Table("Account")]
public class Account
{
    [Key]
    [Column("AccountId")]
    public Guid Id { get; set; }

    .
    .
    .
}

[Table("Owner")]
public class Owner
{
    [Key]
    [Column("OwnerId")]
    public Guid Id { get; set; }

    .
    .
    .
}
```

GetOwnerById GET Request in .NET Core

Modify the `IOwnerRepository` interface:

```
public interface IOwnerRepository
{
    IEnumerable<Owner> GetAllOwners();
    Owner GetOwnerById(Guid ownerId);
}
```

Then implement the interface in `OwnerRepository.cs`:

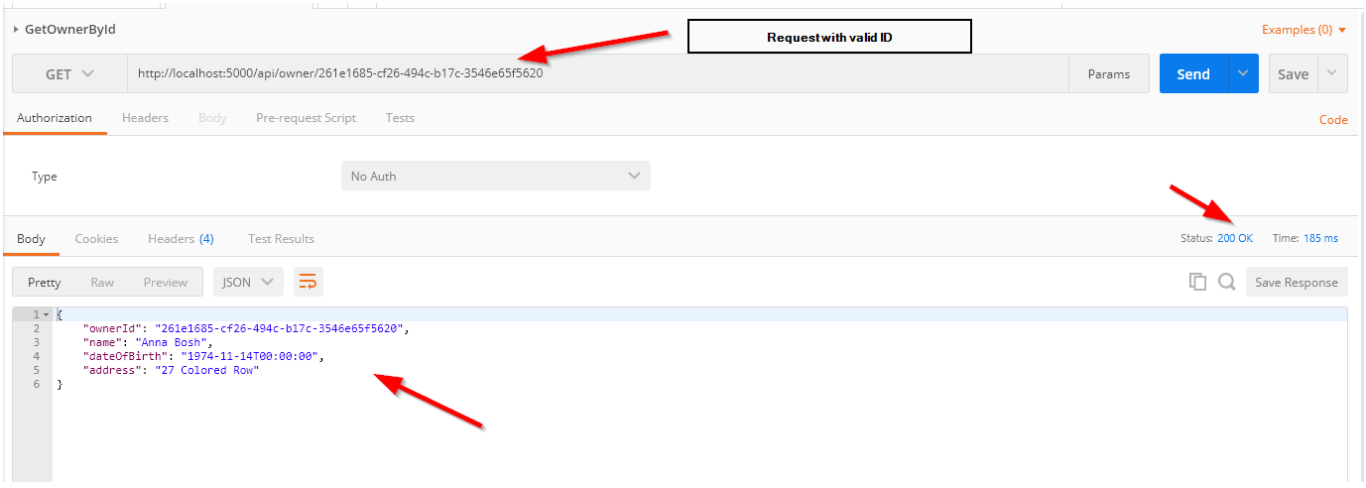
```
public Owner GetOwnerById(Guid ownerId)
{
    return FindByCondition(owner => owner.Id.Equals(ownerId))
            .FirstOrDefault();
}
```

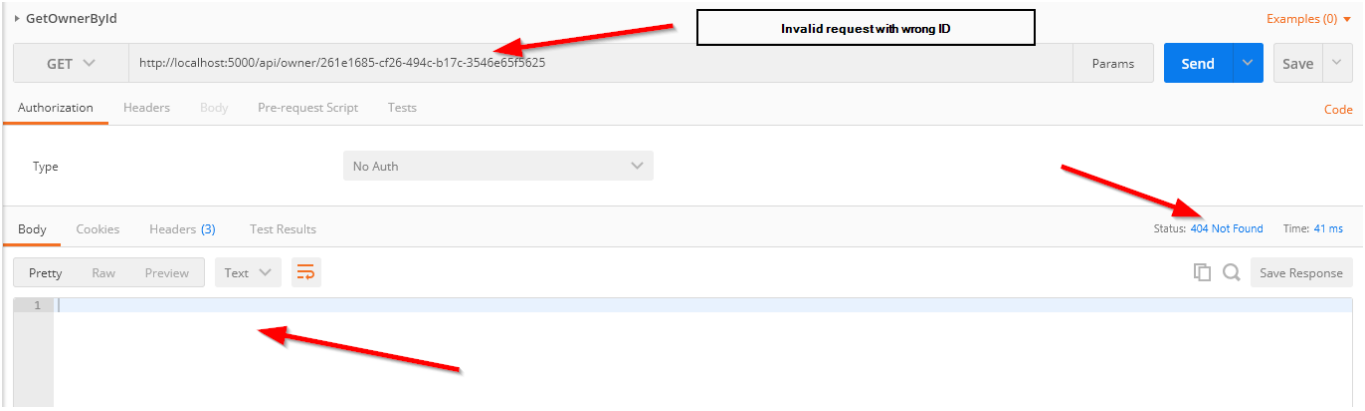
Finally, change the `OwnerController`:

```
[HttpGet("{id}")]
public IActionResult GetOwnerById(Guid id)
{
    try
    {
        var owner = _repository.Owner.GetOwnerById(id);

        if (owner == null)
        {
            _logger.LogError($"Owner with id: {id}, hasn't been found in db.");
            return NotFound();
        }
        else
        {
           _logger.LogInfo($"Returned owner with id: {id}");
           return Ok(owner);
        }
    }
    catch (Exception ex)
    {
        _logger.LogError($"Something went wrong inside GetOwnerById action:
{ex.Message}");
        return StatusCode(500, "Internal server error");
    }
}
```

Use Postman to send valid and invalid requests to check the results:





Code Refactoring

You should try to avoid returning `null` as much as possible because `null` could be the result of many potential bugs in your code. Let's change the code a bit to avoid working with `null`.

First, change the repository method:

```
public Owner GetOwnerById(Guid ownerId)
{
    return FindByCondition(owner => owner.Id.Equals(ownerId))
            .DefaultIfEmpty(new Owner())
            .FirstOrDefault();
}
```

Now we are no longer returning `null` if `ownerId` is invalid, but returning a new `Owner` object with default property values. Returning a new `Owner` object ensures that the OwnerId property will have a default GUID value, which is `Empty.` With that knowledge, we can change `owner == null` to something different:

```
var owner = _repository.Owner.GetOwnerById(id);

if (owner.Id.Equals(Guid.Empty))
{
     _logger.LogError($"Owner with id: {id}, hasn't been found in db.");
     return NotFound();
}
```

DTO and Owner Details Request

Let's create a method to return an owner with its account details.

First, create an extended owner model, or *DTO* (*Data Transfer Object*), which will return the owner with all accounts related to it.

In the `Entities` project, create a new folder called `ExtendedModels,` and inside it create class `OwnerExtended`:

```
public class OwnerExtended
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string Address { get; set; }

    public IEnumerable<Account> Accounts { get; set; }

    public OwnerExtended()
    {
    }

    public OwnerExtended(Owner owner)
    {
        Id = owner.Id;
        Name = owner.Name;
        DateOfBirth = owner.DateOfBirth;
        Address = owner.Address;
    }
}
```

It's a good practice to keep the model classes clean. To add properties to a model class, we can create new extended models. Notice the property `Accounts,` which will bind all the accounts related to the given owner. Also, there is a constructor which takes an owner object as a parameter and maps it to all the properties of this class. All we have to do is to bind all the accounts related to the owner object using the foreign key.

Let’s modify the interface accordingly:

```
public interface IOwnerRepository
{
    IEnumerable<Owner> GetAllOwners();
    Owner GetOwnerById(Guid ownerId);
    OwnerExtended GetOwnerWithDetails(Guid ownerId);
}
```

Also, modify the repository class:

```
public OwnerExtended GetOwnerWithDetails(Guid ownerId)
{
    return new OwnerExtended(GetOwnerById(ownerId))
    {
        Accounts = RepositoryContext.Accounts
            .Where(a => a.OwnerId == ownerId)
    };
}
```

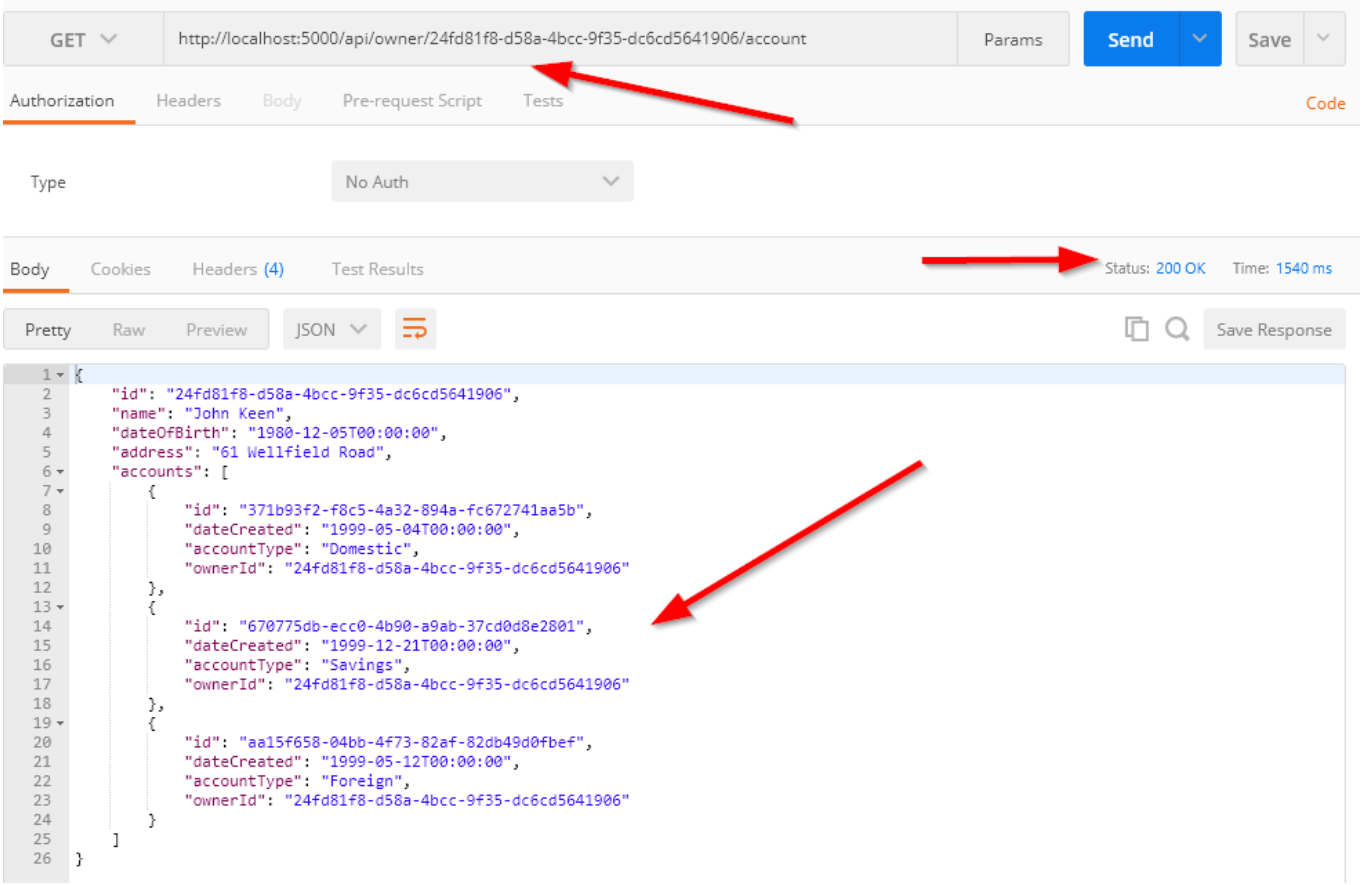
You can see the advantage of removing the null value from the `GetOwnerById` repository method. Because now if you send a wrong `ownerId` value, `GetOwnerById` will return an empty owner object. Thus mapping it into `ownerExtended` won't throw an error as it would if it were `null`.

Finally, modify the controller:

```
[HttpGet("{id}/account")]
public IActionResult GetOwnerWithDetails(Guid id)
{
    try
    {
        var owner = _repository.Owner.GetOwnerWithDetails(id);

        if (owner.Id.Equals(Guid.Empty))
        {
            _logger.LogError($"Owner with id: {id}, hasn't been found in db.");
            return NotFound();
        }
        else
        {
            _logger.LogInfo($"Returned owner with details for id: {id}");
            return Ok(owner);
        }
    }
    catch (Exception ex)
    {
        _logger.LogError($"Something went wrong inside GetOwnerWithDetails action:
{ex.Message}");
        return StatusCode(500, "Internal server error");
    }
}
```

Result:



Conclusion

Lessons learned:

- How to work with a controller class
- What is routing and how to use it
- How to handle GET requests in a web API
- How to use DTOs while handling requests