



# ASP.net REST API

demo code from:  
[Julio Casal](#)



# Introduction

Target Framework: **net5.0**

Entry Point: **Program.cs** (as in all .net apps)



# Startup.cs

`Startup.cs` will be the class invoked by the `Program.cs` Host initiator

- `Configuration property` gets passed in the constructor (used for env files and more)
- `Configure Services` method is used to register `SERVICES` used throughout the API
- `Configure` method is used to manage the request handling pipeline configuration(`MIDDLEWARE`).



# /Controllers

/Controllers/ folder is for classes that handles the ROUTES that the API exposes



# appsettings.json

In the folder `/Properties/` you can create multiple `appsettings` files for settings such as the log level.

Example:

`appsettings.DEVELOPMENT.json`

`appsettings.PRODUCTION.json`

`appsettings.TESTING.json`



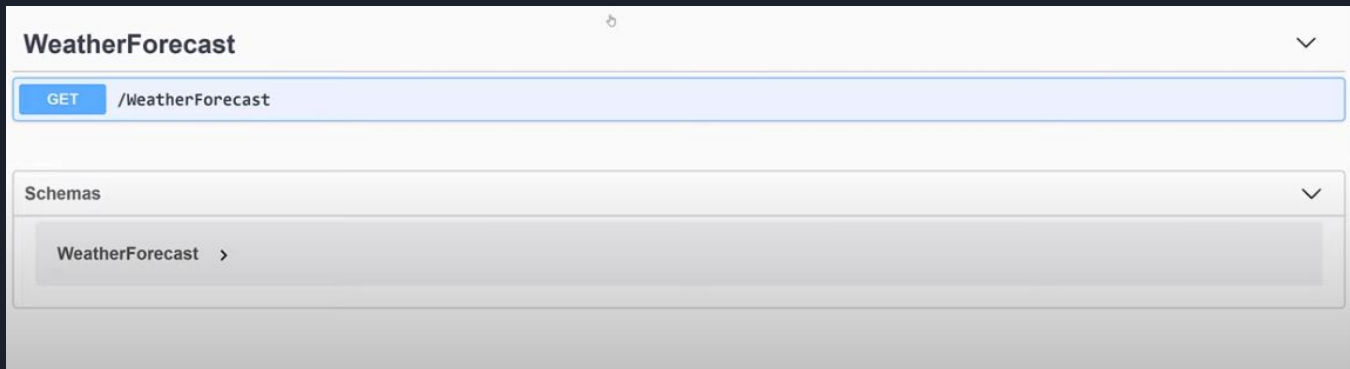
# launchSettings.json

Server configuration such as **application URL** , port numbers , IIS configuration



# Swagger

**Swagger** is an open API specification application that makes your life a lot easier in testing the API.





# Entities - Models

Lets see an example of a **Model** class inside a **/Model/** Folder with usage of **RecordTypes** , and **init** accessor instead of set (c# 9 , net5.0)

```
using System;

namespace Catalog.Entities
{
    public record Item
    {
        public Guid Id { get; init; }
        public string Name { get; init; }
        public decimal Price { get; init; }
        public DateTimeOffset CreatedDate {get; init; }
    }
}
```





# Repositories

The **repository** layer isolates Business layer from the Data Access Layer. The Repository contains Data Mapper entity. This entity can be used as a model entity for providing schema of the data for performing **CRUD** operations, by using the **CRUD** operations defined in the repository.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Catalog.Entities;

namespace Catalog.Repositories
{
    public class InMemItemsRepository
    {
        private readonly List<Item> items = new()
        {
            new Item { Id = Guid.NewGuid(), Name = "Potion", Price = 9, CreatedDate = DateTimeOffset.UtcNow },
            new Item { Id = Guid.NewGuid(), Name = "Iron Sword", Price = 20, CreatedDate = DateTimeOffset.UtcNow },
            new Item { Id = Guid.NewGuid(), Name = "Bronze Shield", Price = 18, CreatedDate = DateTimeOffset.UtcNow }
        };

        public IEnumerable<Item> GetItems()
        {
            return items;
        }

        public Item GetItem(Guid id)
        {
            return items.Where(item => item.Id == id).SingleOrDefault();
        }
    }
}
```

# Example controller class

The following class binds with the **ItemRepository** and the **ItemEntity**

```
using System;
using System.Collections.Generic;
using Catalog.Entities;
using Catalog.Repositories;
using Microsoft.AspNetCore.Mvc;
```

```
namespace Catalog.Controllers
{
    [ApiController]
    [Route("items")]
    public class ItemsController : ControllerBase
    {
        private readonly InMemItemsRepository repository;

        public ItemsController()
        {
            repository = new InMemItemsRepository();
        }
    }
}
```

```
// GET /items
[HttpGet]
public IEnumerable<Item> GetItems()
{
    var items = repository.GetItems();
    return items;
}

// GET /items/{id}
[HttpGet("{id}")]
public ActionResult<Item> GetItem(Guid id)
{
    var item = repository.GetItem(id);

    if (item is null)
    {
        return NotFound();
    }


    return item;
}
}
```



# Dependency Injection

Up until now our code generates a new list of Items every time the Repository class is being called. To avoid that we will inject the repository class in the controller class constructor.

```
public ItemsController()  
{  
    repository = new ItemsRepository();  
}
```



```
public ItemsController(repository)  
{  
    this.repository = repository;  
}
```



# Implementation of injection

We will be using the `IServiceProvider` for this

1. Extract `interface` for `ItemRepository` class
2. Create `interface` class and paste the extracted interface
3. Make sure `repository` implements the `interface`
4. Change repository variable (in `controller`) from `ItemRepository` type to the `interface` type.
5. Receive repository in `Controller` constructor parameters
6. Startup.cs -> ConfigureServices -> `services.AddSingleton<ItemsInterface, ItemsRepository>();`



# DTO - data transfer object

DTOs help to further decouple presentation from the service layer and the domain model. When DTOs are used, the presentation layer and the service layer share data contracts rather than classes. Eventually hiding our models from the outside world.

```
using System;

namespace Catalog.Dtos
{
    public record ItemDto
    {
        public Guid Id { get; init; }
        public string Name { get; init; }
        public decimal Price { get; init; }
        public DateTimeOffset CreatedDate { get; init; }
    }
}
```



# DTO - Extensions.cs

We will create a new root file **Extensions.cs** which will extend the definitions of our types.

**MUST** be **static**!

```
using Catalog.Dtos;
using Catalog.Entities;

namespace Catalog
{
    public static class Extensions
    {
        public static ItemDto AsDto(this Item item)
        {
            return new ItemDto
            {
                Id = item.Id,
                Name = item.Name,
                Price = item.Price,
                CreatedDate = item.CreatedDate
            };
        }
    }
}
```



# DTO - final

No we can actually do the following in the **controller** `getItem` method

```
var items = repository.GetItems().Select(item=> item.AsDto());  
return items;
```

---

And in the `getItem(id)` method

```
var item = repository.GetItem(id);  
  
if (item is null)  
{  
    return NotFound();  
}  
  
return item.AsDto();
```



# Post requests

First we will update our **interface** for the **Repository**

```
using System;
using System.Collections.Generic;
using Catalog.Entities;

namespace Catalog.Repositories
{
    public interface IItemsRepository
    {
        Item GetItem(Guid id);
        IEnumerable<Item> GetItems();
        void CreateItem(Item item);
    }
}
```





# Post request

The next step is to implement the **create** function in the **Repository** class

```
public void CreateItem(Item item)
{
    items.Add(item);
}
```



# Post request - DTO

Lets create a `CreateItemDto` class that will contain only the variables for the `Item` model that we need the user to `POST`. `GUID` and `DATE` will be generated from the `service`.

```
using System.ComponentModel.DataAnnotations;

namespace Catalog.Dtos
{
    public record CreateItemDto
    {
        [Required]
        public string Name { get; init; }

        [Required]
        [Range(1, 1000)]
        public decimal Price { get; init; }
    }
}
```



# Post request

The last step is to update our **Controller**, note that we will return a **location** header.

```
// POST /items
```

```
[HttpPost]
public ActionResult<ItemDto> CreateItem(CreateItemDto itemDto)
{
    Item item = new()
    {
        Id = Guid.NewGuid(),
        Name = itemDto.Name,
        Price = itemDto.Price,
        CreatedDate = DateTimeOffset.UtcNow
    };

    repository.CreateItem(item);

    return CreatedAtAction(nameof(GetItem), new { id = item.Id }, item.AsDto());
}
```



# PUT request - updating items

Add the method in the **Interface** and implement it in the **repository**

```
using System;
using System.Collections.Generic;
using Catalog.Entities;

namespace Catalog.Repositories
{
    public interface IItemsRepository
    {
        Item GetItem(Guid id);
        IEnumerable<Item> GetItems();
        void CreateItem(Item item);
        void UpdateItem(Item item);
    }
}
```

```
public void UpdateItem(Item item)
{
    var index = items.FindIndex(existingItem => existingItem.Id == item.Id);
    items[index] = item;
}
```



# PUT - DTO

Let's create the **DTO** for the **UPDATE** method

```
using System.ComponentModel.DataAnnotations;

namespace Catalog.Dtos
{
    public record UpdateItemDto
    {
        [Required]
        public string Name { get; init; }

        [Required]
        [Range(1, 1000)]
        public decimal Price { get; init; }
    }
}
```



# PUT - update controller class

Finally we will update the **CONTROLLER**

```
// PUT /items/{id}
[HttpPut("{id}")]
public ActionResult UpdateItem(Guid id, UpdateItemDto itemDto)
{
    var existingItem = repository.GetItem(id);

    if (existingItem is null)
    {
        return NotFound();
    }

    Item updatedItem = existingItem with
    {
        Name = itemDto.Name,
        Price = itemDto.Price
    };

    repository.UpdateItem(updatedItem);

    return NoContent();
}
```



# DELETE

Let's implement the **DELETE** request. Update the interface and the **repository**

```
using System;
using System.Collections.Generic;
using Catalog.Entities;

namespace Catalog.Repositories
{
    public interface IItemsRepository
    {
        Item GetItem(Guid id);
        IEnumerable<Item> GetItems();
        void CreateItem(Item item);
        void UpdateItem(Item item);
        void DeleteItem(Guid id);
    }
}
```

```
public void DeleteItem(Guid id)
{
    var index = items.FindIndex(existingItem => existingItem.Id == id);
    items.RemoveAt(index);
}
```



# DELETE

We will not need a new **DTO** for this request so let's just update the **controller**

```
// DELETE /items/{id}
[HttpDelete("{id}")]
public ActionResult DeleteItem(Guid id)
{
    var existingItem = repository.GetItem(id);

    if (existingItem is null)
    {
        return NotFound();
    }

    repository.DeleteItem(id);

    return NoContent();
}
```