CODE MAZE

# ASP.NET CORE WEB API BEST PRACTICES

https://code-maze.com

# TABLE OF CONTENTS

# Introduction

While we are working on a project, our main goal is to make it work as it supposed to and fulfill all the customer's requirements.

But wouldn't you agree that creating a project that works is not enough? Shouldn't that project be maintainable and readable as well?

It turns out that we need to put a lot more attention to our projects to write them in more readable and maintainable way. The main reason behind this statement is that probably we are not the only ones who will work on that project. Other people will most probably work on it once we are done with it.

**So, what should we pay attention to?**

In this guide, we are going to write about what we consider to be the best practices while developing the .NET Core Web API project. How we can make it better and how to make it more maintainable.

So, let's go through some of the best practices we can apply when working with ASP.NET Web API project.

# Startup Class and the Service Configuration

In the `Startup` class, there are two methods: the `ConfigureServices` method for registering the services and the `Configure` method for adding the middleware components to the application's pipeline.

So, the best practice is to keep the `ConfigureServices` method clean and readable as much as possible. Of course, we need to write the code inside

that method to register the services, but we can do that in more readable and maintainable way by using the Extension methods.

For example, let's look at the wrong way to register CORS:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options =>
    {
        options.AddPolicy("CorsPolicy",
            builder => builder.AllowAnyOrigin()
            .AllowAnyMethod()
            .AllowAnyHeader()
            .AllowCredentials());
    });
}
```

Even though this way will work just fine, and will register CORS without any problem, imagine the size of this method after registering dozens of services.

That’s not readable at all.

The better way is to create an extension class with the static method:

```
public static class ServiceExtensions
{
    public static void ConfigureCors(this IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddPolicy("CorsPolicy",
                builder => builder.AllowAnyOrigin()
                .AllowAnyMethod()
                .AllowAnyHeader()
                .AllowCredentials());
        });
    }
}
```

And then just to call this extended method upon the `IServiceCollection` type:

```
public void ConfigureServices(IServiceCollection services)
{
    services.ConfigureCors();
}
```

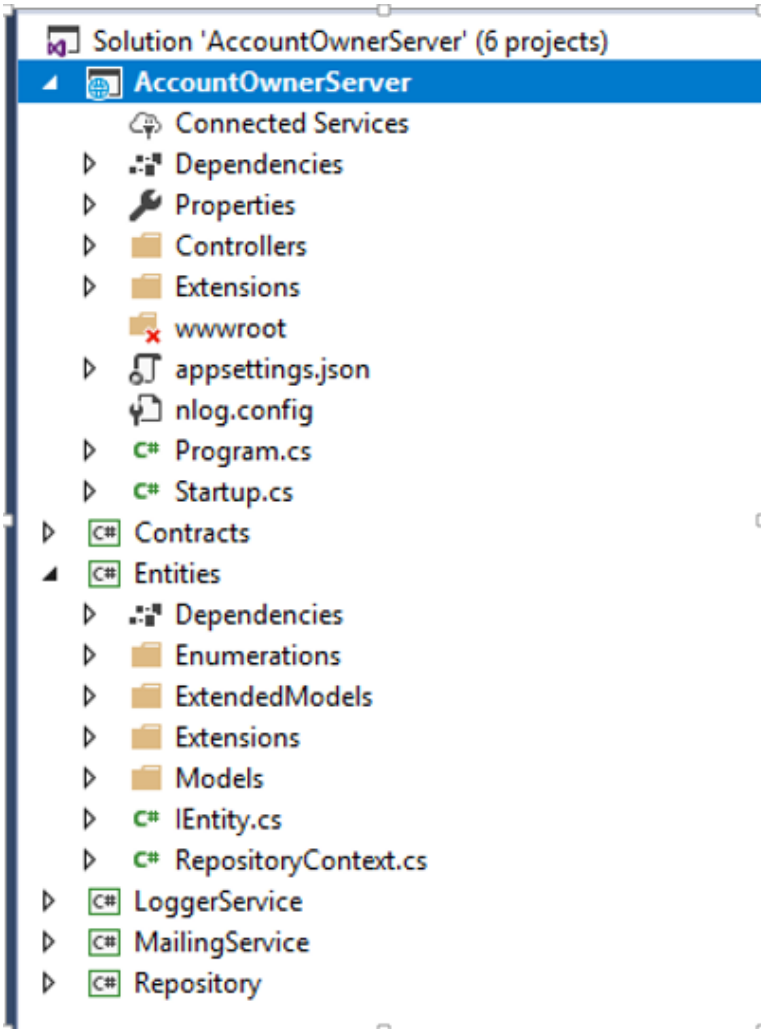
To learn more about the .NET Core's project configuration check out: .NET Core Project Configuration.

## PROJECT ORGANIZATION

We should always try to split our application into smaller projects. That way we are getting the best project organization and separation of concerns (SoC). The business logic related to our entities, contracts, accessing the database, logging messages or sending an email message should always be in a separate .NET Core Class Library project.

Every small project inside our application should contain a number of folders to organize the business logic.

Here is just one simple example how a complete project should look like:

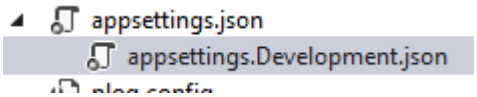



## Environment Based Settings

While we develop our application, that application is in the development environment. But as soon as we publish our application it is going to be in the production environment. Therefore having a separate configuration for each environment is always a good practice.

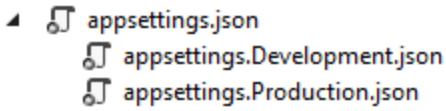
In .NET Core, this is very easy to accomplish.

As soon as we create the project, we are going to get the `appsettings.json` file and when we expand it we are going to see the `appsetings.Development.json` file:




All the settings inside this file are going to be used for the development environment.

We should add another file `appsettings.Production.json`, to use it in a production environment:




The production file is going to be placed right beneath the development one.

With this setup in place, we can store different settings in the different appsettings files, and depending on the environment our application is on, .NET Core will serve us the right settings. For more information about this topic, check out Multiple Environments in ASP.NET Core.

## Data Access Layer

In many examples and different tutorials, we may see the DAL implemented inside the main project and instantiated in every controller. This is something we shouldn't do.

When we work with DAL we should always create it as a separate service. This is very important in the .NET Core project because when we have DAL as a separate service we can register it inside the IOC (Inversion of Control)

container. The IOC is the .NET Core's built-in feature and by registering a DAL as a service inside the IOC we are able to use it in any controller by simple constructor injection:

```
public class OwnerController: Controller
{
    private IRepository _repository;

    public OwnerController(IRepository repository)
    {
        _repository = repository;
    }
}
```

The repository logic should always be based on interfaces and generic as well. Check out this post: .Net Core series – Part 4 to see how we implement the Repository Pattern inside the .NET Core's project.

## CONTROLLERS

The controllers should always be as clean as possible. We shouldn't place any business logic inside it.

So, our controllers should be responsible for accepting the service instances through the constructor injection and for organizing HTTP action methods (GET, POST, PUT, DELETE, PATCH…):

```
public class OwnerController: Controller
{
    private ILoggerManager _logger;
    private IRepository _repository;

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

    [HttpGet]
    public IActionResult GetAllOwners()
    {
    }

    [HttpGet("{id}", Name = "OwnerById")]
    public IActionResult GetOwnerById(Guid id)
    {
    }

    [HttpGet("{id}/account")]
    public IActionResult GetOwnerWithDetails(Guid id)
    {
    }

    [HttpPost]
    public IActionResult CreateOwner([FromBody]Owner owner)
    {
    }

    [HttpPut("{id}")]
    public IActionResult UpdateOwner(Guid id, [FromBody]Owner owner)
    {
    }

    [HttpDelete("{id}")]
    public IActionResult DeleteOwner(Guid id)
    {
    }
}
```

## ACTIONS

Our actions should always be clean and simple. Their responsibilities include handling HTTP requests, validating models, catching errors and returning responses:

```
[HttpPost]
public IActionResult CreateOwner([FromBody]Owner owner)
{
    try
    {
        if (owner.IsObjectNull())
        {
            return BadRequest("Owner object is null");
        }

        if (!ModelState.IsValid)
        {
            return BadRequest("Invalid model object");
        }

        _repository.Owner.CreateOwner(owner);

        return CreatedAtRoute("OwnerById", new { id = owner.Id }, owner);
    }
    catch (Exception ex)
    {
        _logger.LogError($"Something went wrong inside the CreateOwner
action: {ex}");
        return StatusCode(500, "Internal server error");
    }
}
```

Our actions should have `IActionResult` as a return type in most of the cases (sometimes we want to return a specific type or a JsonResult...). That way we can use all the methods inside .NET Core which returns results and the status codes as well.

The most used methods are:

- **OK** => returns the 200 status code
- **NotFound** => returns the 404 status code
- **BadRequest** => returns the 400 status code
- **NoContent** => returns the 204 status code
- **Created**, **CreatedAtRoute**, **CreatedAtAction** => returns the 201 status code
- **Unauthorized** => returns the 401 status code
- **Forbid** => returns the 403 status code
- **StatusCode** => returns the status code we provide as input

## HANDLING ERRORS GLOBALLY

In the example above, our action has its own `try-catch` block. This is very important because we need to handle all the errors (that in another way would be unhandled) in our action method. Many developers are using `try-catch` blocks in their actions and there is absolutely nothing wrong with that approach. But, we want our actions to be clean and simple, therefore, removing `try-catch` blocks from our actions and placing them in one centralized place would be an even better approach.

.NET Core gives us an opportunity to implement exception handling globally with a little effort by using built-in and ready to use middleware. All we have to do is to add that middleware in the `Startup` class by modifying the `Configure` method:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
     app.UseExceptionHandler(config =>
     {
          config.Run(async context =>
          {
               context.Response.StatusCode = 500;
               context.Response.ContentType = "application/json";

               var error = context.Features.Get<IExceptionHandlerFeature>();
               if (error != null)
               {
                    var ex = error.Error;

                    await context.Response.WriteAsync(new ErrorModel()
                    {
                         StatusCode = 500,
                         ErrorMessage = ex.Message
                    }.ToString());
               }
          });
     });

     app.UseMvc();
}
```

We can even write our own custom error handlers by creating custom middleware:

```
public class CustomExceptionMiddleware
{
    //constructor and service injection

    public async Task Invoke(HttpContext httpContext)
    {
        try
        {
            await _next(httpContext);
        }
        catch (Exception ex)
        {
            _logger.LogError("Unhandled exception ...", ex);
            await HandleExceptionAsync(httpContext, ex);
        }
    }
}
```

After that we need to register it and add it to applications pipeline:

```
public static IApplicationBuilder UseCustomExceptionMiddleware(this
IApplicationBuilder builder)
{
    return builder.UseMiddleware<CustomExceptionMiddleware>();
}
app.UseCustomExceptionMiddleware();
```

## USING ACTIONFILTERS TO REMOVE DUPLICATED CODE

Filters in ASP.NET Core allows us to run some code prior to or after the specific stage in a request pipeline. Therefore, we can use them to execute validation actions that we need to repeat in our action methods.

When we handle a PUT or POST request in our action methods, we need to validate our model object as we did in the Actions part of this article. As a result, that would cause the repetition of our validation code, and we want to

avoid that (Basically we want to avoid any code repetition as much as we can).

We can do that by using the ActionFilters. Instead of validation code in our action:

```
if (!ModelState.IsValid)
{
  // bad request and logging logic
}
```

We can create our filter:

```
public class ModelValidationAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        if (!context.ModelState.IsValid)
        {
                context.Result = new
                        BadRequestObjectResult(context.ModelState);
        }
    }
}
```

And register it in the `Startup` class in the `ConfigureServices` method:

```
services.AddScoped<ModelValidationAttribute>();
```

Now, we can use that filter with our action methods.

## MICROSOFT.ASPNETCORE.ALL META-PACKAGE

***NOTE: If you are using the 2.1 and later version of ASP.NET Core it is recommended to use the Microsoft.AspNetCore.App package instead of Microsoft.AspNetCore.All due to security reasons. Furthermore, if we create a new WebAPI project with the 2.1 version, we will automatically get the AspNetCore.App package instead of AspNetCore.All.***

This meta-package contains all of the AspNetCore packages, EntityFrameworkCore packages, SignalR package (from version 2.1) and the supporting packages required for running the framework. It is pretty convenient when starting a new project because we don't have to manually install and reference all the packages we might need.

Of course, your machine needs to have the .NET Core runtime installed on it in order to use the AspNetCore meta-package.

## ROUTING

In the .NET Core Web API projects, we should use Attribute Routing instead of Conventional Routing. That's because Attribute Routing helps us match the route parameter names with the actual parameters inside the action methods. Another reason is the description of the route parameters. It is more readable when we see the parameter with the name "ownerId" than just "id".

We can use the `[Route]` attribute on top of the controller and on top of the action itself:

```
[Route("api/[controller]")]
public class OwnerController: Controller
{
      [Route("{id}")]
      [HttpGet]
      public IActionResult GetOwnerById(Guid id)
      {

      }
}
```

There is another way to create routes for the controller and actions:

```
[Route("api/owner")]
public class OwnerController: Controller
{
      [HttpGet("{id}")]
      public IActionResult GetOwnerById(Guid id)
      {

      }
}
```

There are different opinions which way is better, but we would always recommend the second way, and this is something we always use in our projects.

When we talk about the routing we need to mention the route naming convention. We can use descriptive names for our actions, but for the routes/endpoints, we should use NOUNS and not VERBS.

A few wrong examples:

```
[Route("api/owner")]
public class OwnerController : Controller
{
     [HttpGet("getAllOwners")]
     public IActionResult GetAllOwners()
     {
     }

     [HttpGet("getOwnerById/{id}"]
     public IActionResult GetOwnerById(Guid id)
     {
     }
}
```

A few good examples:

```
[Route("api/owner")]
public class OwnerController : Controller
{
    [HttpGet]
    public IActionResult GetAllOwners()
    {
    }

    [HttpGet("{id}")]
    public IActionResult GetOwnerById(Guid id)
    {
    }
}
```

For the more detailed explanation of the Restful practices checkout: Top REST API Best Practices.

## LOGGING

If we plan to publish our application to production, we should have a logging mechanism in place. Log messages are very helpful when figuring out how our software behaves in a production.

.NET Core has its own logging implementation by using the `ILogger` interface. It is very easy to implement it by using Dependency Injection feature:

```
public class TestController: Controller
{
    private readonly ILogger _logger;

    public TestController(ILogger<TestController> logger)
    {
        _logger = logger;
    }
}
```

Then in our actions, we can utilize various logging levels by using the `_logger` object.

.NET Core supports logging API that works with a variety of logging providers. Therefore, we may use different logging providers to implement our own logging logic inside our project.

The NLog is the great library to use for implementing our own custom logging logic. It is extensible, supports structured logging and very easy to configure. We can log our messages in the console window, files or even database.

To learn more about using this library inside the .NET Core check out: .NET Core series – Logging With NLog.

The Serilog is the great library as well. It fits in with the .NET Core built-in logging system.

## CryptoHelper

We won't talk about how we shouldn’t store the passwords in a database as a plain text and how we need to hash them due to security reasons. That's out of the scope of this guide. There are various hashing algorithms all over the internet, and there are many different and great ways to hash a password.

But if need the library that provides support to the .NET Core's application and that is easy to use, the CryptoHelper is quite a good library.

The CryptoHelper is standalone password hasher for .NET Core that uses a PBKDF2 implementation. The passwords are hashed using the new Data Protection stack.

This library is available for installation through the NuGet and its usage is quite simple:

```
using CryptoHelper;

// Method for hashing the password
public string HashPassword(string password)
{
    return Crypto.HashPassword(password);
}

// Method to verify the password hash against the given password
public bool VerifyPassword(string hash, string password)
{
    return Crypto.VerifyHashedPassword(hash, password);
}
```

## Content Negotiation

By default .NET Core Web API returns a JSON formatted result. In most of the cases, that's all we need.

But what if the consumer of our Web API wants another response format, like XML for example?

For that, we need to create a server configuration to format our response in the desired way:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(config =>
    {
        // Add XML Content Negotiation
        config.RespectBrowserAcceptHeader = true;
        config.InputFormatters.Add(new XmlSerializerInputFormatter());
        config.OutputFormatters.Add(new XmlSerializerOutputFormatter());
    });
}
```

Sometimes the client may request a format that is not supported by our Web API and then the best practice is to respond with the status code 406 Not Acceptable. That can be configured inside our `ConfigureServices` method as well:

```
config.ReturnHttpNotAcceptable = true;
```

We can also create our own custom format rules.

The content negotiation is a pretty big topic so if you want to learn more about it, check out: Content Negotiation in .NET Core.

## USING JWT

JSON Web Tokens (JWT) are becoming more popular by the day in the web development. It is very easy to implement JWT Authentication is very easy to implement due to the .NET Core's built-in support. JWT is an open standard and it allows us to transmit the data between a client and a server as a JSON object in a secure way.

We can configure the JWT Authentication in the `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
     services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
       .AddJwtBearer(options =>
       {
          options.TokenValidationParameters = new TokenValidationParameters
          {
             //Configuration in here
          };
       });
}
```

In order to use it inside the application, we need to invoke this code in the `Configure` method:

```
app.UseAuthentication();
```

We may use JWT for the Authorization part as well, by simply adding the role claims to the JWT configuration.

To learn in more detail about JWT authentication and authorization in .NET Core, check out JWT with .NET Core and Angular Part 1 and Part 2 of the series.

## CONCLUSION

In this guide, our main goal was to familiarize you with the best practices when developing a Web API project in .NET Core. Some of those could be used in other frameworks as well, therefore, having them in mind is always helpful.

Thank you for reading the guide and I hope you found something useful in it.