ASP.NET Core App structure

The overall structure of a typical ASP.NET Core app entry point, which typically consists of six steps (mentioned in Program.cs file):

1. Create a WebApplicationBuilder instance.

2. Register the required services and configuration with the WebApplicationBuilder.

3. Call Build() on the builder instance to create a WebApplication instance.

4. Add middleware to the WebApplication to create a pipeline.

5. Map the endpoints in your application.

6. Call Run() on the WebApplication to start the server and handle requests.

Definition (Services and SRP/DI/IoC or SoC)

Within the context of ASP.Net Core, **service** refers to any class that provides functionality to an application. Services could be classes exposed by a library or code you've written for your application.

The single-responsibility principle (SRP) states that every class should be responsible for only a single piece of functionality; it should need to change only if that required functionality changes. SRP is one of the five main design principles promoted by Robert C. Martin in Agile Software Development, Principles, Patterns, and Practices (Pearson, 2013).

This technique is called dependency injection or the Inversion of Control (IoC) principle, a well-recognized design pattern that is used extensively. Typically, you'll register the dependencies of your application into a container, which you can use to create any service. You can use the container to create both your own custom application services and the framework services used by ASP.NET Core. You must register each service with the container before using it in your application.

Endpoints

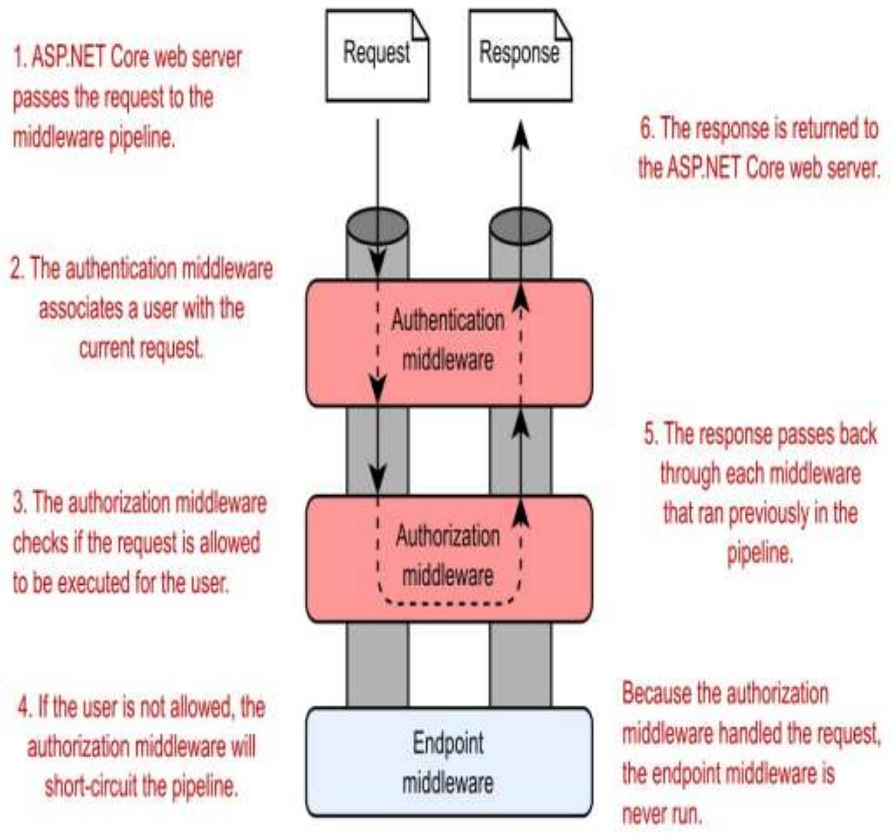
After registering your services with the IoC container on WebApplicationBuilder and doing any further customization, you create a WebApplication instance. You can do three main things with the WebApplication instance:

- Add middleware to the pipeline.
- Map endpoints that generate a response for a request.
- Run the application by calling Run().

Middleware pipeline

This arrangement—whereby a piece of middleware can call another piece of middleware, which in turn can call another, and so on—is referred to as a pipeline. You can think of each piece of middleware as being like a section of pipe; when you connect all the sections, a request flows through one piece and into the next. One of the most common use cases for middleware is for the cross-cutting concerns of your application. These aspects of your application need to occur for every request, regardless of the specific path in the request or the resource requested, including:

- Logging each request
- Adding standard security headers to the response
- Associating a request with the relevant user
- Setting the language for the current request


1. ASP.NET Core web server passes the request to the middleware pipeline.
Request
Response
6. The response is returned to the ASP.NET Core web server.
2. The authentication middleware associates a user with the current request.
Authentication middleware
5. The response passes back through each middleware that ran previously in the pipeline.
3. The authorization middleware checks if the request is allowed to be executed for the user.
Authorization middleware
4. If the user is not allowed, the authorization middleware will short-circuit the pipeline.
Endpoint middleware
Because the authorization middleware handled the request, the endpoint middleware is never run.

3.8 Summary

- The .csproj file contains the details of how to build your project, including which NuGet packages it depends on. Visual Studio and the .NET CLI use this file to build your application.
- Restoring the NuGet packages for an ASP.NET Core application downloads all your project's dependencies so that it can be built and run.
- Program.cs is where you define the code that runs when your app starts. You can create a `WebApplicationBuilder` by using `WebApplication.CreateBuilder()` and call methods on the builder to create your application.
- All services, both framework and custom application services, must be registered with the `WebApplicationBuilder` by means of the `Services` property, to be accessed later in your application.
- After your services are configured you call `Build()` on the `WebApplicationBuilder` instance to create a `WebApplication` instance. You use `WebApplication` to configure your app's middleware pipeline, to register the endpoints, and to start the server listening for requests.
- Middleware defines how your application responds to requests. The order in which middleware is registered defines the final order of the middleware pipeline for the application.
- The `WebApplication` instance automatically adds `RoutingMiddleware` to the start of the middleware pipeline and `EndpointMiddleware` as the last middleware in the pipeline.
- Endpoints define how a response should be generated for a given request and are typically tied to a request's path. With minimal APIs, a simple function is used to generate a response.
- You can start the web server and begin accepting HTTP requests by calling `Run` on the `WebApplication` instance.