<Coding𝄞onata />

A QUICK GUIDE TO LEARN ASP.NET CORE WEB API

Aram Tchekrekjian

@AramT87

Introduction

- ASP.NET Core Web API is the major upgrade over ASP.NET Web API
- Can be easily deployed and run under different platforms
- Based on Modules to allow extensibility through NuGet Packages and custom middleware
- Dependency Injection is at the core of the framework, so no need to add DI libraries

.NET Updates

.NET 8 is now the latest major release with a Long Term Support (LTS) until November, 2026

C# 12 is the latest version of C# that comes bundled with .NET 8

ASP.NET Core Web API Components & Topics

@AramT87

ASP.NET Core Web API Components

- Controllers
- Routing
- Versioning
- Middleware
- Attributes
- Filters
- Program.cs
- appsettings.json
- Dependency Injection
- Kestrel

@AramT87

Controllers & Routing

- Controllers define the HTTP methods and routing URLs (endpoints) that will be exposed to the public (clients).

- When a client sends an HTTP request to our API, ASP.NET Core will map the request with the specified route and forward the request to the matching controller

- In ASP.NET Core Web API, Controllers inherit from ControllerBase.

- The ControllerBase has methods and properties to help the developer handle the HTTP requests/responses. For example there are many methods that wrap your response with a specific type of http status, such as:

return Ok(productModel);

Versioning

- You can define multiple versions of the same API or endpoints by introducing the versioning prefix

- Versioning is important to allow for adding or modifying the API without affecting the existing clients

- You can use ApiVersion Attribute to specify the version you are currently working with

- To use ApiVersion, you will have to import the Microsoft.AspNetCore.Mvc.Versioning NuGet Package

- You can also map a specific action to Api Version using the MapToApiVersion Attribute

Middleware

- A piece of Software class or component that can be injected somewhere in the pipeline of the ASP.NET Core application

- The purpose to handle or process request/response

- Middleware can vary from predefined ones like UseAuthentication() or Obtained from NuGet like UseSwagger()

- Or custom built like a middleware class to intercept the request and check for ‘api-key’ header in the request headers and authenticate the client accordingly

Attributes

- Reusable decorations(data annotations) added to class, method or property to validate data before or after processing the request

- In ASP.NET Core, you will be working with different attributes mainly on Controller level, such as [Route("..."), [ApiController], [HttpGet] and more

- You can also build your own attribute by creating a class and inheriting from System.Attribute Class

- Attributes can implement different number of Filters to handle different tasks

- A good example for an attribute is ApiKeyAttribute, where you can use it on a specific Controller to make it authenticated via api-key header

Filters

- Filters are used to intercept the processing pipeline, so you can run some code before or after some stage in the process

- A good example is a filter exception to handle and log exceptions globally in your API project instead of adding try catch blocks into every controller

- When combined together, Attribute allows filters to accept arguments.

- In the ASP.NET filter pipeline, there are 5 types of filters:
  - Authorization Filters
  - Resource Filters
  - Action Filters
  - Exception Filters
  - Result Filters

Program.cs

- This is the entry point to your application, from here your application (before it is even an ASP.NET Core) starts.

- It includes the initial setup for services and configuration for your whole Web API project.

- In Program.cs class you will define an instance of WebApplication.CreateBuilder(args), from which you will read the Services Collection, Configuration and other important components

- Later you will build the WebApplication and configure the different middleware to be used in your pipeline

appsettings.json

- This is a JSON formatted file that includes all the environment-based settings and configurations that you can use in ASP.NET Core Web API

- appsettings.json file usually include ConnectionStrings, AppSettings Key,Value pairs like URLs to other APIs and services, Logging Configurations...etc.

- Using the Options Pattern, the appsettings content can be easily accessed from different components of your ASP.NET Core project through strongly-typed class that represent your appsettings structure, this happens by injecting IOptions<> into your different classes

Dependency Injection

- This is a key topic in ASP.NET Core sitting at the core of the ASP.NET Core framework

- ASP.NET Core provides built-in service container to handle all the dependency registration

- With a single line of code, dependency can be easily defined in program.cs using the different Add Methods inside the ServiceCollection found in the WebApplicationBuilder's instance

- Dependencies added into the service collection can be defined by 3 types of lifetimes:

Dependency Lifetimes

Transient

- New instance of the dependency would be created on each HTTP request
- Also a new instance would be created upon each request for dependency within the same HTTP Request
- No issues with thread safety

Dependency Lifetimes

Scoped

- Instance would be shared within the same HTTP request
- An instance is created upon the initial request for a dependency and any subsequent request for a dependency would share the same instance.
- Good usage example for scoped is the DbContext of Entity Framework Core
- No issues with thread safety

Dependency Lifetimes

Singleton

- Instance would be shared across all the different HTTP requests

- You can use Singleton to share data across multiple HTTP requests such as application-level caching data

- Thread-safety might be an issue here since a Singleton can be accessed from multiple HTTP requests where each request would be running under its own thread(s)

Kestrel

- The default web server that comes bundled with ASP.NET Core
- Allows ASP.NET Core applications to be deployed cross-platform
- It is lightweight and much faster than other web servers, since it doesn't provide all the features of other servers like hosting static files, URL rewriting, compression...etc.
- It must be run behind a reverse-proxy server

ASP.NET Core Web API More Topics

- Web API Security
- Exception Handling
- Logging
- Entity Framework Core
- File Upload
- Swagger OpenAPI
- Localization
- DTO & AutoMapper
- Fluent API Validations
- Deployment
- Testing

Thank You

Follow me for more content

Aram Tchekrekjian

AramT87

CodingSonata.com/newsletters

<Coding♪onata />