Contents

[Application Architecuture 2](#_Toc107170149)

[Creating Web API Project and Exploring default files 3](#_Toc107170150)

[launchSettings.json 3](#_Toc107170151)

[.csproj file 4](#_Toc107170152)

[Converting an Console Core Application to Web API project 5](#_Toc107170153)

[Project .csproj file level changes 5](#_Toc107170154)

[Program.cs file level changes 5](#_Toc107170155)

[Startup.cs file level changes 5](#_Toc107170156)

[Host Builder 6](#_Toc107170157)

[CreateDefaultBuilder 6](#_Toc107170158)

[ConfigureWebHostDefaults 6](#_Toc107170159)

[Startup Class 6](#_Toc107170160)

[Core Concepts 6](#_Toc107170161)

[Controller 6](#_Toc107170162)

[Middleware and HTTP Request Pipeline 7](#_Toc107170163)

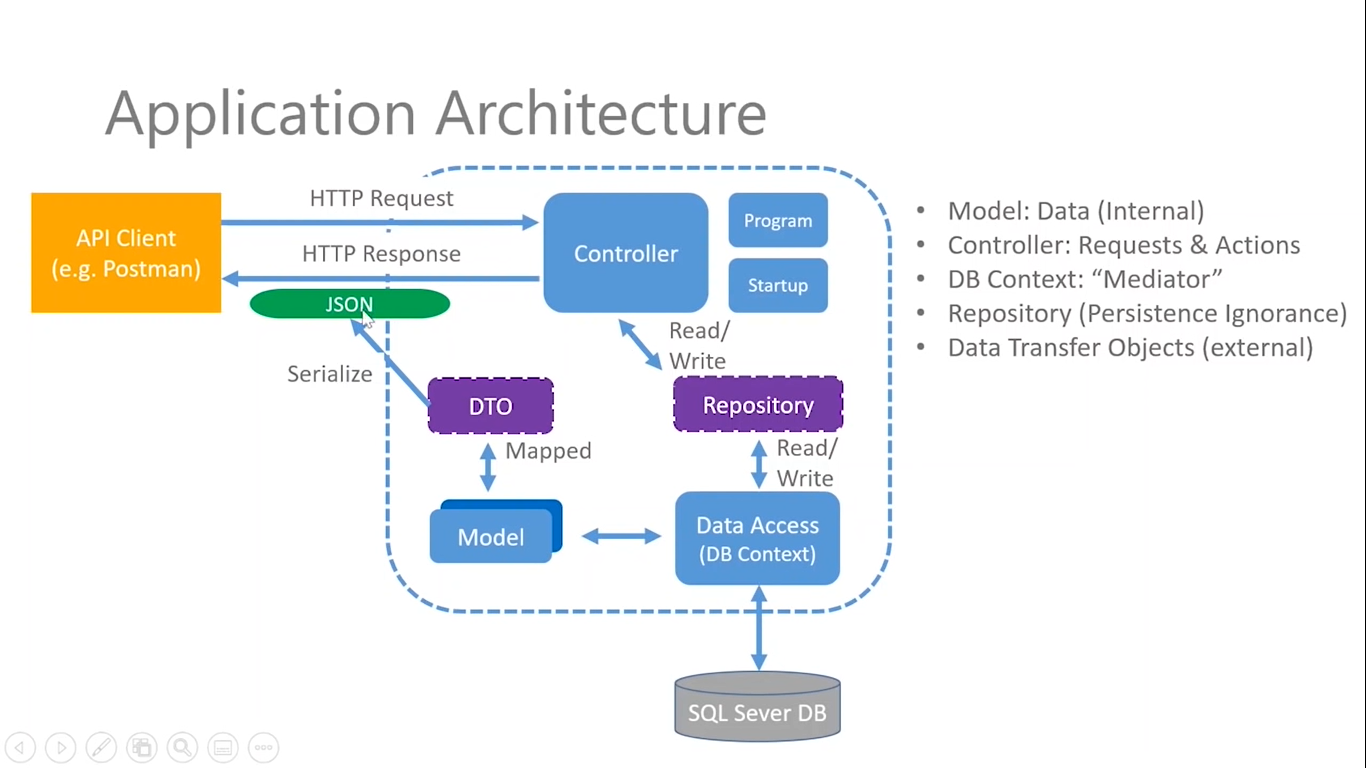
[Http Request Pipeline 8](#_Toc107170164)

[Middleware 8](#_Toc107170165)

[Middleware methods 8](#_Toc107170166)

[Implementing Custom Middleware 10](#_Toc107170167)

# Application Architecuture



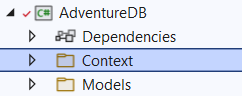
**Entity Framework Core**

|  |
| --- |
| PM> Scaffold-DbContext "Data Source=DESKTOP-PD9SE9U\SQLEXPRESS01;Initial Catalog=AdventureWorks2019;Integrated Security=True" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -ContextDir Context -Context AdventureContext -force |

Replacing above hard coded connection string with Appsettings.json connection

|  |
| --- |
| "ConnectionStrings": {  "AdventureDb": "Data Source=DESKTOP-PD9SE9U\\SQLEXPRESS01;Initial Catalog=AdventureWorks2019;Integrated Security=True"  }, |

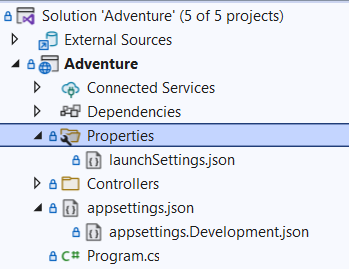
|  |
| --- |
| Scaffold-DbContext -Connection Name=AdventureDb Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -ContextDir Context -Context AdventureContext -Force |



# Creating Web API Project and Exploring default files

Creating a web api project. Below is the default project structure will get created

Let’s discuss about one by one



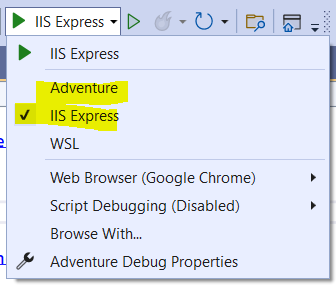
## launchSettings.json

|  |
| --- |
| {  "$schema": "https://json.schemastore.org/launchsettings.json",  "iisSettings": {  "windowsAuthentication": false,  "anonymousAuthentication": true,  "iisExpress": {  "applicationUrl": "http://localhost:46176",  "sslPort": 44369  }  },  "profiles": {  "Adventure": {  "commandName": "Project",  "dotnetRunMessages": true,  "launchBrowser": true,  "launchUrl": "swagger",  "applicationUrl": "https://localhost:7209;http://localhost:5209",  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Development"  }  },  "IIS Express": {  "commandName": "IISExpress",  "launchBrowser": true,  "launchUrl": "swagger",  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Development"  }  }  }  } |

As .net core is capable of running on any OS, above configuration will useful to run the web api.

If it is running on windows application, system will use IIS express setting, if it is running on other OS, system will use kestrel web server.

We can switch the web server using below



## .csproj file

When we double click on project, it will open a file call .csproj file (other way to open is Right Click on project and select Edit project file). It will gives the information about

* Targeting SDK
* Packages installed
* Project references and other project specific details

|  |
| --- |
| <Project Sdk="Microsoft.NET.Sdk.Web">  <PropertyGroup>  <TargetFramework>net6.0</TargetFramework>  <Nullable>enable</Nullable>  <ImplicitUsings>enable</ImplicitUsings>  <UserSecretsId>sdfdsf-ewrwe-rtytu-</UserSecretsId>  </PropertyGroup>  <ItemGroup>  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.5">  <PrivateAssets>all</PrivateAssets>  <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>  </PackageReference>  <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />  </ItemGroup>  <ItemGroup>  <ProjectReference Include="..\AdventureRepository\AdventureRepository.csproj" />  <ProjectReference Include="..\AdventureService\AdventureService.csproj" />  </ItemGroup>  </Project> |

# Converting an Console Core Application to Web API project

## Project .csproj file level changes

* Change the Project SDK to web
* Remove the Output Type
* Verify the target framework moniker (TFM)

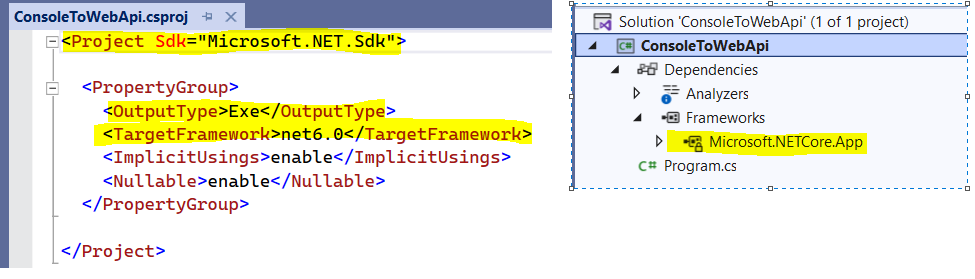
## Program.cs file level changes

* Add the Web Host Builder
* Configure the startup class

## Startup.cs file level changes

* Add Routing
* Set Default Route

Step#1: Below is the default project.config file and framework from console app



Step#2: lets modify the project sdk, TargetFramework and remove the OutypeType file.

After modification will observe another framework will get added with the name Microsoft.ASPNETCore.App



Program.cs file changes

Modified Class

|  |
| --- |
| static void Main(String[] arrgs)  {  CreateHostBuilder(arrgs);  }  public static IHostBuilder CreateHostBuilder(String[] arrgs) =>  Host.CreateDefaultBuilder(arrgs).ConfigureWebHostDefaults(webBuilder =>  {  webBuilder.UseStartup<Startup>();  }); |

### Host Builder

Host Builder is an object that is used to add some default features in the application

### CreateDefaultBuilder

Below are some of the responsibilities of CreateDefaultBuilder

* Enable scope validation on the **Dependency Injection (DI)** container
* Set the **ContentRootPath** to the result of System.IO **GetCurrentDirectory**
* Load app configuration from **appsettings.json,command line args, Environment variables**
* Load app configuration from **User Secrets** when environment set to “**Development**”
* Configure the **ILoggerFactory** to **log** the console, debug and event source output

### ConfigureWebHostDefaults

* Provides support for HTTP
* Use Kestrel as the web server and configure it using the application configurations providers
* Enables the IIS integration
* Adds the HostFiltering middleware
* Adds the ForwardedHeaders middleware if ASPNETCORE\_FORWARDEDHEADER\_ENABLED=true

### Startup Class

* Startup class have two important methods
  + ConfigureServices
  + Configure
* ConfigureServices: This method will take IServiceCollection as an parameter, which is responsible for registering the services(Contract)
* Configure: This method will take IApplicationBuilder and IWebHostEnvironment as a paramters, which is responsible for configuring the application requests (pipeline) and provide the information about web hosting environment in which the application is running.

# Core Concepts

## Controller

* A controller class in webapi has a “controller” suffix
* The controller class must inherited from **ControllerBase**
  + **ControllerBase** class providers basic methods and properties to handle HTTP request.
* Use **ApiController** attribute on the controller, it is responsible for
  + Attribute Routing requirements
  + Handle the client errors like 400 status code etc
  + Multipart/Form data request interface
  + Bind the incoming data with the parameters using some more attributes
* Use Attribute Routing
  + Attribute Routing helps us to define route specific to method

Code sample

|  |
| --- |
| [ApiController]  [Route("test/[action]")]  public class TestController : ControllerBase  {  public string Get()  {  return "Hello From Get";  }  public string Get1()  {  return "Hello From Get1";  }  } |

Urls:

<https://localhost:44369/test/get>

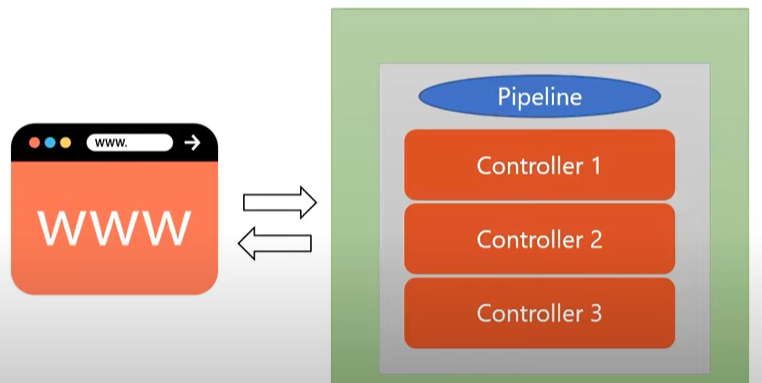
<https://localhost:44369/test/get1>

# Middleware and HTTP Request Pipeline

**Background**: let’s understand the basic flow, when we request a specific url from browser.

In general, we assume that whenever we make a request to the server specific url, a particular controller action method will invoke and processed with output.

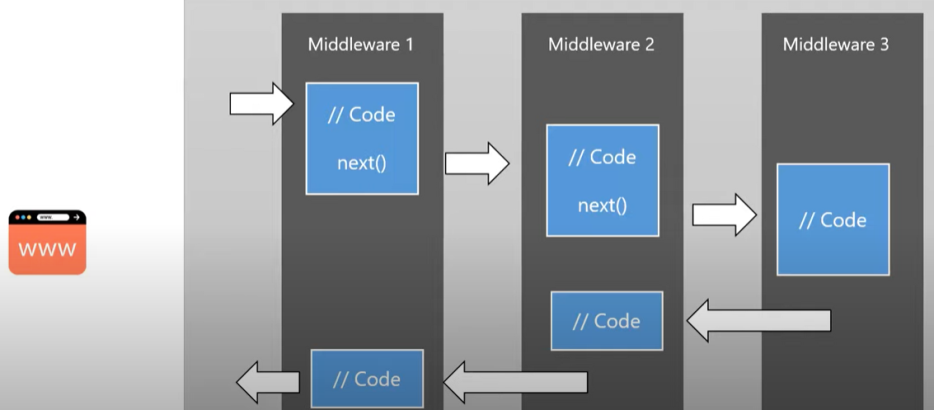
But in reality, we have the Pipeline for Http Requests.



## Http Request Pipeline

We need to configure the middleware in the respective order of their execution, so middleware will internally call the other middlewares. Upon completing the configuration pipeline the middleware will run.

Note: The order of execution is matters a lot.



## Middleware

* Middleware is the piece of code that is used in the Http request pipeline
* An Asp.net core web application can have “N” number of middleware
* Order of execution matters a lot during execution

Examples of some predefined middleware are Routing, Authentication, Add Error Page etc

### Middleware methods

**Run()**

This method is used to **complete** the middleware execution

**Microsoft - Adds a terminal middleware delegate to the application request pipeline**

Prefer using the Run() as below for better performance

|  |
| --- |
| app.Use((context, next) =>  // {  // return next(context);  // }); |

**Use()**

This method is used to **Insert a new** middleware in the pipeline

Microsoft – Adds a middleware delegate define in-line to the application request pipeline

**Next()**

This method is used to **pass** the execution to the next middleware

* Map(): This method is used to **map** the middleware to the specific URL

Program.cs (in VS2022) Startup.cs (below VS2022)

**Run() Demo**

|  |
| --- |
| app.Run(async option =>  {  await option.Response.WriteAsync("Hello from Run");  });  app.Run(async option =>  {  await option.Response.WriteAsync("Hello from Run 2");  }); |

Output: Hello from Run

Explanation: As said above, Run() method will complete/terminate the application request pipeline. Hence, system will not go for next configured middleware after Run() executed.

**Use() & Next() Demo**

|  |
| --- |
| app.Use(async (context, next) =>  {  await context.Response.WriteAsync("Hello from Use 1 \n");  await next();  });    app.Run(async option =>  {  await option.Response.WriteAsync("Hello from Run \n");  }); |

Output:

Hello from Use 1

Hello from Run

Case Study#1

|  |
| --- |
| app.Use(async (context, next) =>  {  await context.Response.WriteAsync("Hello from Use 1-1 \n");  await next();  await context.Response.WriteAsync("Hello from Use 1-2 \n");  });  app.Run(async option =>  {  await option.Response.WriteAsync("Hello from Run \n");  }); |

Output:

Hello from Use 1-1

Hello from Run

Hello from Use 1-2

Case Study#2

|  |
| --- |
| app.Use(async (context, next) =>  {  await context.Response.WriteAsync("Hello from Use 1-1 \n");  await next();  await context.Response.WriteAsync("Hello from Use 1-2 \n");  });  app.Use(async (context, next) =>  {  await context.Response.WriteAsync("Hello from Use 2-1 \n");  await next();  await context.Response.WriteAsync("Hello from Use 2-2 \n");  });  app.Run(async option =>  {  await option.Response.WriteAsync("Hello from Run \n");  }); |

Output:

Hello from Use 1-1

Hello from Use 2-1

Hello from Run

Hello from Use 2-2

Hello from Use 1-2

Explanation

As discussed in above Http Request Pipeline, Use() method is used to add the middleware, next() is use to pass execution to next middleware and Run() is use to complete middleware execution.

In the above use case #1,#2. System added middleware in order for Use 1-1 & 1-2 (by calling next())

Once system is complete the Run() execution, then control is travelling back from Use 2-2 & 2-1. Please refer above Http Request Pipeline for execution flow.

# Implementing Custom Middleware

Below is the step to create a custom middleware

Step#1: Create a class which Implement **IMiddleware**

Step#2: Provide implementation to the methods of IMiddleware interface

Step#3: Dependency Injection (DI) add services to the container (in ConfigureService() method)

Step#4: Configure Middle in Http Request pipeline (in Configure method using .UseMiddleware())

Step#1 & #2

|  |
| --- |
| public class CustomDemoMiddleware : IMiddleware  {  public async Task InvokeAsync(HttpContext context, RequestDelegate next)  {  await context.Response.WriteAsync("Custom: Hello from Use 2-1 \n");  await next(context);  await context.Response.WriteAsync("Custom: Hello from Use 2-2 \n");  }  } |

Step#3

|  |
| --- |
| builder.Services.AddTransient<CustomDemoMiddleware>(); |

Step#4

|  |
| --- |
| app.UseMiddleware<CustomDemoMiddleware>(); |