.NET Core

Introduction

THIS DOCUMENT COVERS

- Introduction

.NET Core Framework is a cross platform subset of the .NET Framework. It ships with a new runtime known as Core CLR that supports IL compilation and garbage collection, but which does not support app domains. The class libraries are contained in fine grained packages. At present the .NET Core Framework can only be used for creating ASP.NET and console applications. One major difference is that .NET Framework core can be deployed side by side with an application.

CLI

Everything needed to build, test, run .net core applications is provided from the command line via the .NET Core Command-line Interface CLI. On windows it lives in

```
C:\Program Files\dotnet.
```

To run CLI commands we use a tool known as the driver (dotnet.exe). Some useful commands are

Show version of driver	`dotnet --version`
List installed SDK versions	`dotnet --list-sdks`
List Installed Runtime versions	`dotnet --list-runtimes`
List templates	`dotnet new --list`
Build .net project	`dotnet build`

If one does not want to use the latest installed version of the CLI we need to create a `global.json` file at the root level of the application, we are building. This file looks as follows.

```
{
    "sdk": {
       "2.0.0"
    }
}
```

> Note: This only determines the version of the CLI tools used. It does not affect the version of the runtime used by the application. The version of the runtime is specified by the `TargetFramework` element of the `.csproj` file.

Command Line Applications

Simple Command Line

The simplest kind of .NET core application is the humble command line executable. Consider the following setup which consists of three files.

1. Properties/launchSettings.json
2. CommandLineApp.csproj
3. Program.cs

Listing 1CommandLineApp.csproj

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <RootNamespace>_1._CommandLineApp</RootNamespace>
  </PropertyGroup>
</Project>
```

Listing 2 Program.cs

```
class Program
{
    static void Main(string[] args)
    {
          WriteLine(args[0]);
          WriteLine(Environment.GetEnvironmentVariable("EnvVar"));
    }
}
```

Listing 3 Properties/launchSettings.json

```
{
  "profiles": {
    "1. CommandLineApp": {
      "commandName": "Project",
      "commandLineArgs": "AnArg=Hello",
      "environmentVariables": {
        "EnvVar": "World"
      }
    }
  }
}
```

From the directory that contains the .csproj file we can run the application with the relevant arguments and environment variables from the command line as follows.

```
dotnet run --verbosity normal --launchProfile Project
```

Simple Hosted Service Command Line

In .NET Core a host encapsulates the resources required by a running application such as

1. Environment
2. Configuration
3. Logging
4. Dependency Injection
5. Lifecycle management

Typically, the host is built, created, and ran from inside `Program.Main`. The `Host.CreateDefaultBuilder()` builds a particularly useful host can be used in many scenarios. The following code shows a simple hosted application that uses `CreateDefaultBuilder`. This is the next level of complexity from the previous section and indeed we see this in the structure of the project.

Listing 4 GenericHost.csproj

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <RootNamespace>_2._GenericHost</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="5.0.0" />
    <PackageReference Include="Microsoft.Extensions.Hosting.Abstractions" Version="5.0.0" />
  </ItemGroup>

  <ItemGroup>
    <None Update="appsettings.development.json">
      <CopyToOutputDirectory>Always</CopyToOutputDirectory>
    </None>
    <None Update="appsettings.json">
      <CopyToOutputDirectory>Always</CopyToOutputDirectory>
    </None>
  </ItemGroup>

</Project>
```

Listing 5 Properties/launchSettings.json

```
{
  "profiles": {
    "2. GenericHost": {
      "commandName": "Project",
      "environmentVariables": {
        "DOTNET_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Listing 6 appsettings.json

```
{
  "SettingOne" :  "Hello Generic Host"
}
```

Listing 7 appsettings.development.json

```
{
  "SettingOne" :  "Development Env Overriden Value"
}
```

Listing 8 Program.cs

```
class Program
{
      static void Main(string[] args)
      {
            CreateHostBuilder()
                  .ConfigureServices(collection =>
                        collection.AddHostedService<MyService>())
                  .Build()
                  .Run();
      }

      public static IHostBuilder CreateHostBuilder() => Host.CreateDefaultBuilder();
}

public class MyService : IHostedService
{
      public MyService(IConfiguration configuration) =>
            WriteLine(configuration["SettingOne"]);
      public Task StartAsync(CancellationToken cancellationToken) => Task.CompletedTask;
      public Task StopAsync(CancellationToken cancellationToken) => Task.CompletedTask;
}
```

Customizing host configuration

If we can manually recreate what `CreateDefaultBuilder` doeswe will better understand the .NET host builder and be able to use it to create customized behaviour over and above `CreateDefaultBuilder`. Basically, `CreateDefaultBuilder` deals with initializing.

- Host Environment
- Application Configuration
- Logging

We will deal with each one in turn.

HOST ENVIRONMENT

The host environment is encapsulated by the type `IHostEnvironment` which has three properties.

- ApplicationName
- EnvironmentName
- ContentRootPath

The extension method `ConfigureHostConfiguration` allows us to define where the application will look for host environment settings. What is more, the order in which sources are defined is important. Values from source defined later can override values for the same key from sources defined earlier. Consider the following code. It looks for environment variables with the prefix `DOTNET_` and finally looks for command line arguments.

```
class Program
{
   static void Main(string[] args)
   {
        // Create the host builder
       IHostBuilder hostBuilder = CreateHostBuilder(args)
               .ConfigureServices(collection =>
                       collection.AddHostedService<MyHostedService>());

        // Build the host
       IHost host = hostBuilder.Build();

       // Run the host
       host.Run();
   }

   public static IHostBuilder CreateHostBuilder(string[] args)
   {
        IHostBuilder hostBuilder = new HostBuilder()
               .ConfigureHostConfiguration(builder =>
            AddCustomHostConfiguration(builder, args));

        return hostBuilder;
   }

   public static void AddCustomHostConfiguration(IConfigurationBuilder
   configurationBuilder, string[] args)
   {
        configurationBuilder.AddEnvironmentVariables("DOTNET_");
        configurationBuilder.AddCommandLine(args);
   }
```

By specifying this order in our `AddCustomHostConfiguration` method we ensure that any keys specified as command line arguments override any keys specified i as environment variables. So, see how it all comes together consider the following setup.

Listing 9 Properties/launchsettings.json

```
{
  "profiles": {
    "3. ConfigureCustomHostBuilding": {
      "commandName": "Project",
      "commandLineArgs": "ApplicationName=KennysApp",
      "environmentVariables": {
        "DOTNET_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Then the resulting host environment is as follows. The application name comes from the command line and the environment comes from the environment variable. At runtime, the `Environment` is Development and the `ApplicationName` is KennysApp.

Application Configuration

Once the host environment is configured the next thing to build is the application configuration. The order is intentional. We often want the hosting environment to influence how we load the application configuration. Depending on the value of the environment name or content root we can load different application configuration. The following code is added to the previous section to set up the application configuration in the same way as `CreateDefaultBuilder` would.

```
private static void AddCustomApplicationConfiguration(HostBuilderContext context,
IConfigurationBuilder builder, string[] args)
{
     var hostingEnvironment = context.HostingEnvironment;
     builder.AddJsonFile("appsettings.json", optional: true);
     builder.AddJsonFile($"appsettings.{hostingEnvironment.EnvironmentName}.json");

     if (context.HostingEnvironment.IsDevelopment())
     {
            builder.AddUserSecrets<Program>();
     }

     builder.AddEnvironmentVariables("DOTNET_");
     builder.AddCommandLine(args);
}
```

The host will look for configuration in the following places in the following order.

1. A file called `appsettings.json`
2. A file called `appsettings.{EnvironmentName}.json`
3. If in development environment the user secrets file
4. Any environment variables prefixed with `DOTNET_`
5. Any command line arguments

In the following example the environment is development. In this case at runtime the value of the setting Location is “Remote”.

Listing 10 launchsettings.json

```
{
  "Location" :  "Remote"
}
```

Listing 11 launchsettings.development.json

```
{
  "Location": "Local"
}
```

LOGGING

The final important task the `CreateDefaultBuilder` carries out is to initialize logging. We add a method as follows.

```
private static void AddCustomHostConfiguration(
    IConfigurationBuilder configurationBuilder,
    string[] args)
{
    configurationBuilder.AddEnvironmentVariables("DOTNET_");
    configurationBuilder.AddCommandLine(args);
}
```

The `CreateDefaultBuilder` also adds EventLog logging on windows but we do not show that here.

Dependency Injection (DI)

Dependency Injection is an invaluable tool that helps us build loosely coupled software. Typically in order to instantiate a specific object which we refer to as the root we will have to provide it with a dependency graph of other objects. The DI container creates instances of objects by first creating or locating instances of all its dependencies and passing them into the objects constructor. This is turn requires creates the dependencies of the dependencies and so on hence the term dependency graph.

DI containers usually call the objects they create services which is a bit misleading as they create any objects. .NET Core only supports constructor injection out of the box. Setting up a DI container is known as registration. We register services in the `ConfigureServices` extension method of `IHostBuilder`.

```
CreateHostBuilder()
    .ConfigureServices(collection =>
    {
          collection.AddSingleton<IHello, Hello>();
          collection.AddHostedService<MyService>();
    })
    .Build()
    .Run();
```

Lifetime

The lifetime of an object can be singleton, transient or scoped. A captured dependency occurs when you inject a scoped object into a singleton object. Although it is only supposed to live for the lifetime of the request in ASP.NET it will end up hanging around because of the singleton.

Logging

The basic .NET logging in initialized out of the box if we choose `Host.CreateDefaultBuilder()`. We just add a dependency to our constructor and if our object is created by DI we will obtain a relevant logger.

```
public class MyService : IHostedService
{
    public MyService(IConfiguration configuration, ILogger<MyService> logger) =>
        logger.LogInformation("Constructed");
```

Serilog

Serilog is a powerful third-party logging application that supports structured logging. To enable it we add a dependency on `Serilog.AspNetCore`. Then we add custom configuration of the host as follows.

```
class Program
{
    static void Main(string[] args)
    {
        CreateHostBuilder()
            .ConfigureServices(collection =>
            {
                collection.AddHostedService<MyService>();
            })
            .ConfigureLogging((context, builder) =>
            {
                Log.Logger = new LoggerConfiguration().
                    ReadFrom.Configuration(context.Configuration)
                    .CreateLogger();
            })
            .UseSerilog()
            .Build()
            .Run();
    }

    public static IHostBuilder CreateHostBuilder() => Host.CreateDefaultBuilder();
}
```

Finally, we need to add the configuration to `appsettings.json`

```
{
  "Serilog": {

    "MinimumLevel": {
      "Default": "Information",
      "Override" : {
        "Microsoft": "Information",
        "System" :  "Warning"
      }
     },

    "WriteTo": [
      { "Name": "Console" },
      {
        "Name": "File",
        "Args": { "path": "Logs/log.txt" }
      }
    ],
    "Enrich": [ "FromLogContext", "WithMachineName", "WithThreadId" ]
  }
}
```

ASP.NET Core

Server

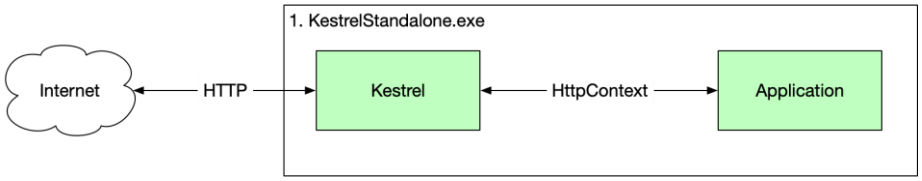
An ASP.NET Core application contains an in-process HTTP Server which listens for HTTP requests and passes them to the application code as a HTTPContext object. All platforms (Linux, MacOS and Windows) ship with Kestrel, which is a high performance, cross platform HTTP Server. If Kestrel is used as the HTTP Server it can either directly server clients or it can sit behind a reverse proxy such as IIS, NGINX or Apache.

In addition to Kestrel, Windows also ships with two other in-process HTTP servers.

- IIS Server – in-process server for IIS
- HTTP.sys server is based on HTTP.sys kernel driver and HTTP Server API

Neither of these tow servers work in reverse proxy configuration. For the rest of this article, we will focus on Windows. In addition, we will use a Visual Studio development environment to run each of the different configurations possible on Windows.

KESTREL BY ITSELF



We can set this up using the following code.

> **HTTPS**
>
> In this example we expose HTTP and HTTPS endpoints with the default being HTTPS.

First, we set the HTTP server to be Kestrel using the `UseKestrel` method in the Program.cs.

Listing 12 Program.cs

```
public class Program
{
    public static void Main(string[] args)
    {
          var host = new WebHostBuilder()
                .UseKestrel()
                .Configure(appBuilder => appBuilder.Run(async ctx =>
                      await ctx.Response.WriteAsync("Hello World")))
                .Build();

          host.Run();
    }
}
```

Secondly, we set the value of the `commandName` property in our `launchSettings.json` file to be Project which causes dotnet to run this projects executable as a standalone process.

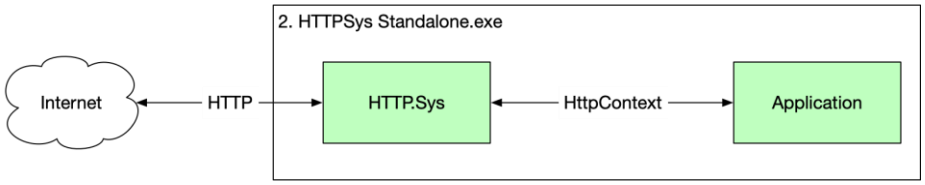
Listing 13 Properties/launchsettings.json

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "KestrelStandalone": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "https://localhost:5000;http://localhost:5001"
    }
  }
}
```

At runtime we have only one process `1.KestrelStandalone.exe`

The full source code for this project can be found here Source Code

HTTP.SYS BY ITSELF



> **HTTPS**
>
> In this example we only expose HTTP as I do not want to go through the pain of setting up HTTPS in my development environment.

First, we set the HTTP server to be Kestrel using the `UseKestrel` method in the Program.cs.

Listing 14 Program.cs

```
public class Program
{
      public static void Main(string[] args)
      {
            var host = new WebHostBuilder()
                  .UseHttpSys()
                  .Configure(appBuilder => appBuilder.Run(
                        async ctx =>
                              await ctx.Response.WriteAsync("Hello World")))
                  .Build();

            host.Run();
      }
}
```

Secondly, we set the value of the `commandName` property in our `launchSettings.json` file to be `Project` which causes dotnet to run this projects executable as a standalone process.

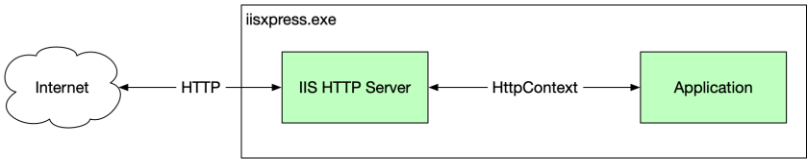
Listing 15 Program/launchSettings.json

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "HttpSysStandalone": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000"
    }
  }
}
```

At runtime we have only one process `2. HTTPSys Standalone.exe`

The full source code for this project can be found here [Source Code]

IIS IN PROCESS



> **HTTPS**
>
> In this example we expose both http and http endpoints

First, we set our `Program.cs` to use IIS.

Listing 16 Program.cs

```
public class Program
{
     public static void Main(string[] args)
     {
            var host = new WebHostBuilder()
                   .UseIIS()
                   .Configure(appBuilder => appBuilder.Run(async ctx =>
                          await ctx.Response.WriteAsync("Hello World")))
                   .Build();

            host.Run();
     }
}
```

Secondly, we set the value of the `commandName` property in our `launchSettings.json` file to be `IISExpress` which causes dotnet to run this projects executable as a standalone process.

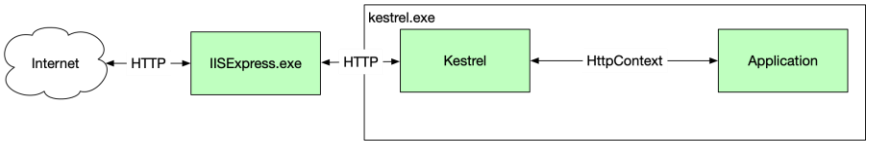
Listing 17 Properties/launchSettings.json

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:57647",
      "sslPort": 44360
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

We now no longer have a .NET executable running. We only have the IIS process named `IISExpress.exe`

The full source code for this project can be found here Source Code

IIS OUT OF PROCESS



> **HTTPS**
>
> In this example we expose both http and http endpoints on http://localhost:57647/ and https://localhost:44360

First, we set our `Program.cs` to use Kestrel.

Listing 18 Program.cs

```
public class Program
{
     public static void Main(string[] args)
     {
            var host = new WebHostBuilder()
                   .UseKestrel()
                   .Configure(appBuilder => appBuilder.Run(async ctx =>
                          await ctx.Response.WriteAsync("Hello World")))
                   .Build();

            host.Run();
     }
}
```

Secondly, we set the value of the `commandName` property in our `launchSettings.json` file to be `IISExpress` which causes dotnet to run this projects executable as a standalone process. Furthermore, we add the `ancmHostingModel` and set it to `OutOfProcess`.

Listing 19 Properties/launchSettings.json

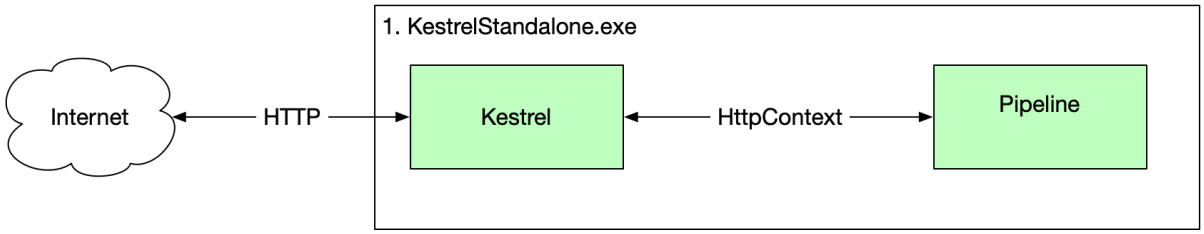
```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:57696",
      "sslPort": 44337
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "ancmHostingModel": "OutOfProcess"
    }
  }
}
```

Now when we run our app we have two processes. Our .NET process called `4. IIS Out Of Process.exe` and `iisexpress.exe`

The source code is here Source Code

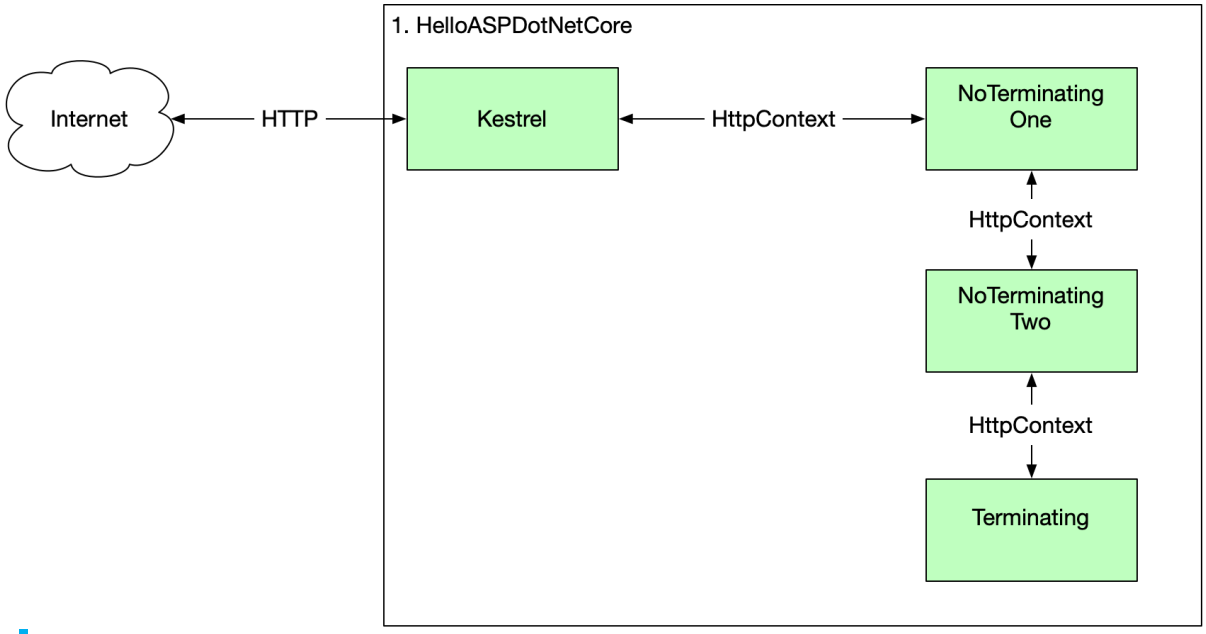
Pipeline

We can visualize the ASP.NET core architecture as the following.



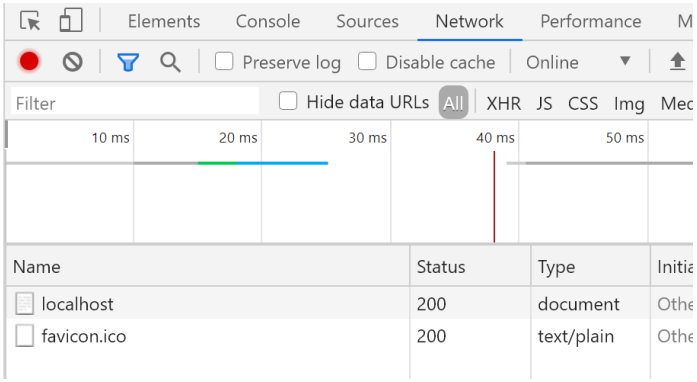
The pipeline consists of zero or more non-terminating middleware components, following by a single terminating middleware component. The non-terminating middleware components can pre and post-process the request and response. The terminating middleware is the actual logic. Consider the following code.

```
public class Program
{
	public static void Main(string[] args)
	{
		var host = new WebHostBuilder()
			.UseKestrel()
			.Configure(ConfigureApp)
			.Build();

		host.Run();
	}

	private static void ConfigureApp(IApplicationBuilder applicationBuilder)
	{
		//applicationBuilder.Use(NonTerminatingOne);
		//applicationBuilder.Use(NonTerminatingTwo);
		applicationBuilder.Run(TerminatingMiddleware);
	}

	private static async Task NonTerminatingOne(HttpContext httpContext,
		Func<Task> next)
	{
		// Pre-processing
		WriteLine("NonTerminating One - Preprocessing");

		// Actual task
		await next();

		// Post Processing
		WriteLine("NonTerminating One - Postprocessing");
	}

	private static async Task NonTerminatingTwo(HttpContext httpContext,
		Func<Task> next)
	{
		// Pre-processing
		WriteLine("NonTerminating Two - Preprocessing");

		// Actual task
		await next();

		// Post Processing
		WriteLine("NonTerminating Two - Postprocessing");
	}

	private static async Task TerminatingMiddleware(HttpContext httpContext)
	{
		await httpContext.Response.WriteAsync("Hello World");
		WriteLine("Terminating Done");
	}
}
```



MY CODE IS CALLED TWICE

You might notice your pipeline is called twice from the browser. This is because the browser is asking for the fav icon as well as the main resource. I see this.



For a nice diagram of the order of standard .NET core middleware see

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-5.0

HTTPS And Certificates

HTTPS And Development Certificates

I have set up a super simple ASP.NET Core project that simply returns "Hello World". In this project I have deliberately removed all IIS information from `launchsettings.json`.

Listing 20 01.HelloHTTPS.csproj

```
<Project Sdk="Microsoft.NET.Sdk.Web">
    <PropertyGroup>
    <TargetFramework>netcoreapp5.0</TargetFramework>
    <RootNamespace>_1._HelloHTTPS</RootNamespace>
  </PropertyGroup>
</Project>
```

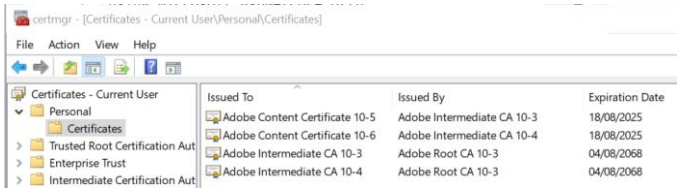
Listing 21 Program.cs

```
    public class Program
    {
        public static void Main(string[] args)
        {
            new WebHostBuilder()
                .UseKestrel()
                .Configure(bld=>bld.Run(async ctx =>
                    await ctx.Response.WriteAsync("Hello World")))
                .Build()
                .Run();
        }
    }
```

Listing 22 Properties/launchSettings.json

```
{
  "profiles": {
    "01.HelloHTTPS": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "https://localhost:5001;http://localhost:5000"
    }
  }
}
```

I have also removed the localhost certificate from the following location so there is no certificate.

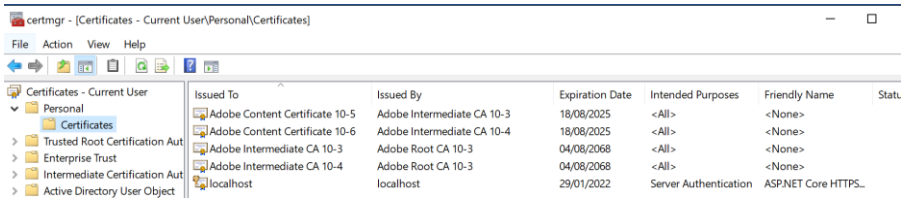


Now when I run the application, I see the following error telling me we have no certificate.

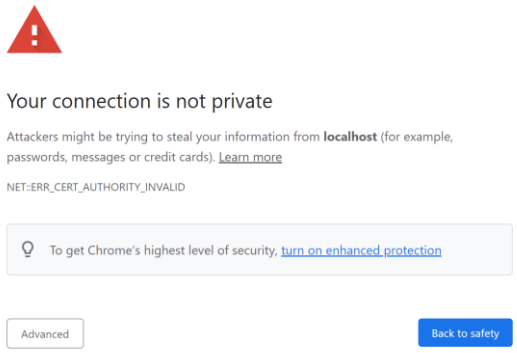
Let us go ahead and create a developer certificate as instructed.

```
C:\Users\rps>dotnet dev-certs https
The HTTPS developer certificate was generated successful
```

Let us go ahead and create a developer certificate as instructed. If we **restart** the certificate manager, we see.



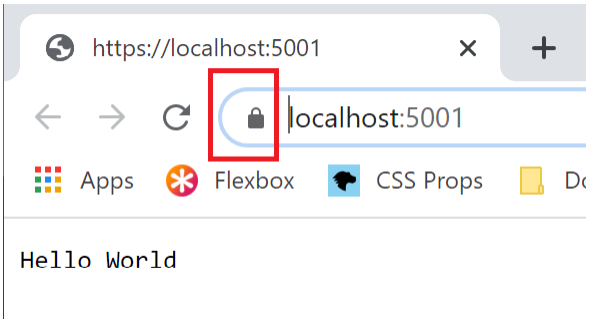
Now when we start out app from Visual Studio we see the following.



We need to trust the certificate. We should have used the following form of the command to create the certificate and trust it.

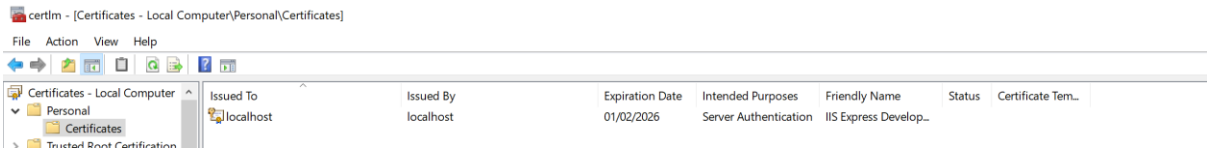
```
dotnet dev-certs https --trust
```

If we select yes .NET will install a certificate and trust it. Now when we run our application, we see the familiar lock in chrome.



IIS Development Certificate

Note that the IIS development certificate can be found in the computer level certificates



Authentication

In this section we look at three ways of authenticating in windows. In all the examples the actual messages have been truncated and replaces with XXX… to protect the credentials.

NTLM

Let us set up a simple project that highlights the NTLM protocol when used between a browser and our HTTP server. First let us define a controller with an authorized action. For the full source code see.

https://bitbucket.org/Riskandpricingsolutions/aspdotnetcorebasics/src/master/security/02.%20Authenticate%20NTLM/

With our project running and listening on the http://localhost:5000/api/hello/message we can use the curl command from a command prompt to view the round trips need to authenticate the client to the server.

```
curl -v -u: --ntlm  http://localhost:5000/api/hello/message
```

The results will highlight for us the authentication handshake which is carried out using HTTP headers encoded in Base64.

> **HTTP.SYS AND NEGOTIATE**
>
> It seems when we use the HTTP.sys end server in ASP.NET core that only the final message is seen in the pipeline. I am guessing the HTTP.sys layer is taking care of the NTML handshake.

1. Client sends negotiation message to server.

The encoded message contains the host and domain. Notice in bold the encode message in the Authorization header.

```
GET /api/hello/message HTTP/1.1
Host: localhost:5000
Authorization: NTLM TlRMTVNXXXX….
User-Agent: curl/7.55.1
Accept: */*
```

2. Server responds with challenge.

The server responds with the challenge which is a random 8-byte number again encoded in Base64.

```
HTTP/1.1 401 Unauthorized
Content-Type: text/html; charset=us-ascii
Server: Microsoft-HTTPAPI/2.0
WWW-Authenticate: NTLM TlRMXXXXXXX…
Date: Fri, 12 Feb 2021 08:14:11 GMT
Content-Length: 341
```

3. Client encrypts.

The client must now encrypt the challenge using the user's credentials to prove it has then. It sends the encrypted value back.

```
GET /api/hello/message HTTP/1.1
```

```
Host: localhost:5000
Authorization: NTLM TlRMTVNTSABDAAAAAAAA…
User-Agent: curl/7.55.1
Accept: */*
```

4. Server Checks

The server checks the result against the one it obtained using the users credentials and returns the document if the values match.

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: text/plain; charset=utf-8
Server: Microsoft-HTTPAPI/2.0
Date: Fri, 12 Feb 2021 08:14:11 GMT

Hello Kenny*
```

So, we can see there are two round trips between the client and the service. This is expected with NTLM. For details see

https://docs.microsoft.com/en-us/windows/win32/secauthn/microsoft-ntlm#:~:text=Windows%20Challenge%2FResponse%20(NTLM),and%20on%20stand%2Dalone%20systems.&text=NTLM%20uses%20an%20encrypted%20challenge,user's%20password%20over%20the%20wire.

NEGOTIATE

We should not explicitly specify NTLM or Kerberos. If specify negotiate the protocol will try and use Kerberos and fall back onto NTML. The following code is an example. Note in this example we are using the Kestrel HTTP server and not HTTP.sys as we did in the previous example.

https://bitbucket.org/Riskandpricingsolutions/aspdotnetcorebasics/src/master/security/03%20Authenticate%20Negotiate/

We use the curl command

```
curl -v -u: --negotiate  http://localhost:5000/api/hello/message
```

The output is then as follows.

1. Client sends message to server.

```
GET /api/hello/message HTTP/1.1
Host: localhost:5000
User-Agent: curl/7.55.1
Accept: */*
```

2. Server responds telling server to use Negotiate.

```
HTTP/1.1 401 Unauthorized
Date: Fri, 12 Feb 2021 08:43:23 GMT
Server: Kestrel
Content-Length: 0
WWW-Authenticate: Negotiate
```

3. Client Sends ???

```
GET /api/hello/message HTTP/1.1
Host: localhost:5000
Authorization: Negotiate YIGXXXX…
  User-Agent: curl/7.55.1
Accept: */*
```

4. Server sends

```
HTTP/1.1 401 Unauthorized
Date: Fri, 12 Feb 2021 08:45:36 GMT
Server: Kestrel
Content-Length: 0
WWW-Authenticate: Negotiate oYIBCzXXX…
```

5. Client Sends

```
GET /api/hello/message HTTP/1.1
Host: localhost:5000
Authorization: Negotiate oXcwXXX…
User-Agent: curl/7.55.1
Accept: */*
```

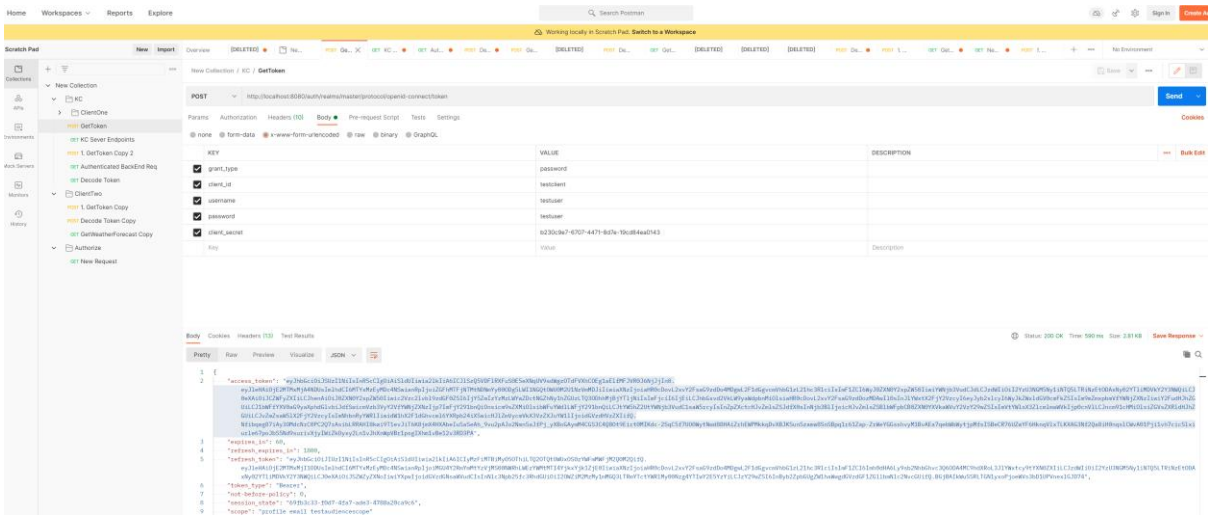
5. Server Sends

```
HTTP/1.1 200 OK
Date: Fri, 12 Feb 2021 08:47:25 GMT
Content-Type: text/plain; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked
WWW-Authenticate: Negotiate oRswGaADCgEAoxIEEAEAAADswe4CxIMi+gAAAAA=

Hello Kenny* Closing connection 0
```

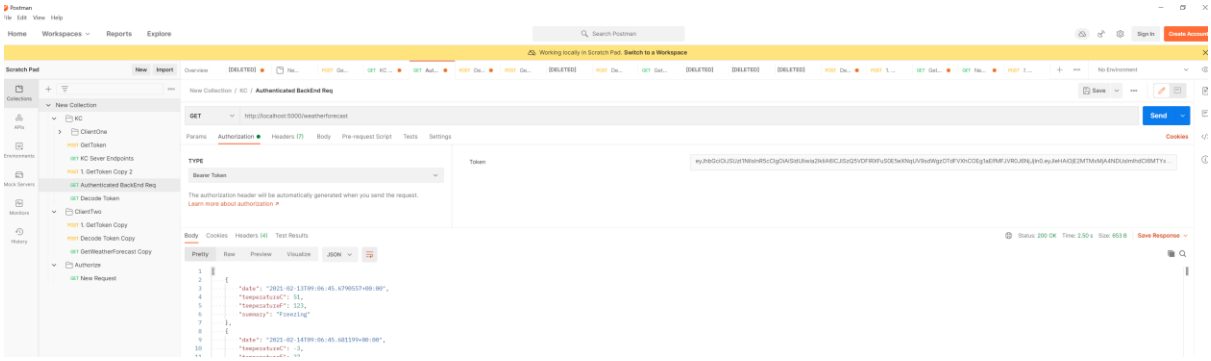
KeyCloak / JWT

This code assumes we have setup a KeyCloak client as specified in my KeyCloak document. We setup the code as per the following link.

Now we execute the protected endpoint using PostMan. First we need to get the KeyCloak bearer token



We then cut and paste the access token into the bearer section in our request.



Authorization

Authorization typically follows authentication. Let us work through some examples. We will use our KeyCloak token as it is easier to insert claims into it that when using windows principals.

Policies

We control authorization using policies which we setup in the ConfigureServices method of Startup.cs. The key parts of authentication in ASP.NET Core are

- Policies
- Restrictions
- Handlers

DEFAULT POLICY

If decorate our endpoint action with the Authorize attribute but give it no string argument it will use the default policy. The code lives in the repository

XXXX

The points of note are.

- Because we don't specify a policy name argument to the Authorize attribute on the action we link to the default policy setup in ConfigureServices
- The policy's only restriction is that the user must be authenticated.

Figure 1 WeatherForecastController.cs

```
[HttpGet]
[Authorize]
public IEnumerable<WeatherForecast> Get()
{
  var rng = new Random();
  return Enumerable.Range(1, 5).Select(index => new WeatherForecast
  {
      Date = DateTime.Now.AddDays(index),
      TemperatureC = rng.Next(-20, 55),
      Summary = Summaries[rng.Next(Summaries.Length)]
  })
  .ToArray();
}
```

CUSTOM POLICY

Listing 23 Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
	services.AddCors();

	services.AddControllers();

	var auth = services.AddAuthentication();

	auth.AddJwtBearer("my_authentication_scheme", options =>
	{
		options.Authority = "http://localhost:8080/auth/realms/master";
		options.Audience = "testclient";
		options.RequireHttpsMetadata = false;
	});

	services.AddAuthorization(options =>
	{
		options.DefaultPolicy = new AuthorizationPolicyBuilder()
			.AddAuthenticationSchemes(new[] {
  "my_authentication_scheme", })
			.RequireAuthenticatedUser()
			.Build();
	});
}
```

NON-DEFAULT POLICY

Let us add a non-default policy.

MVC

The following shows the simplest MVC implementation.

Model Binding

Model binding takes the request and uses it to create the arguments to action methods. Model binding takes values from the following parts of the incoming request

- Form Values
- Route Values
- Query String Values

In addition, action arguments can come from dependency injection.

The `HTTPContext` object encapsulates and represents a single HTTP Request. Kestrel populates this from the actual request and passes it to the rest of the application. Kestrel then hands the context object to the middleware pipeline which performs tasks such as

Authentication

The Kestrel HTTP server creates an `HTTPContext` object for each request it receives. The `User` property of HTPContext object stores the current principle. The principle is the user of the web application. The type of the principle in ASP.Net.Core is `ClaimsPrincipal`. Each ClaimsPrinciple has a set of `Claims` associated with it. Each claim describes a single piece of information and consists of a claim type and an optional value. The following shows some typical claims

Email	Kenny@gmail.com
HasAdminAccess	

Sign in works as follows in a tradition ASP.NET Core Web application

1. User sends login details (Identifier such as email address and secret such as password)
2. Initially the `HTTPContexts` user is set to an anonymous, unauthenticated user
3. The action directs to the `SignInManager` which loads the user from the DB and validates their password

4. If the password is correct the user is signed in and the `HTTPContext.User` property is set to an authenticated user principal.
5. The principal is serialized and returned to the browser as an encrypted cookie.

> **COOKIES**
>
> A cookie is a small piece of text that is sent back and forth between the browser and the app with each request. Each cookie has a name and a value.

Browsers automatically send cookies with all requests made from ones app. ASP.NET Core uses the authentication cookie sent with the requests to rehydrate the ClaimsPrinciple and set the HTTPContext.User principal for the request. This process happens in AuthenticationMiddleware.

At first Kestrel assigns a generic, anonymous, unauthenticated principle with no claims.

The pipeline is a chain of modules that can pre and post process incoming HTTP requests.

Performance

Tiered Compilation

Doing JIT compilation involves compromises. Using aggressive optimisations for every method leads to great steady state performance at the expense of longer start up time. Simpler compilation leads to faster start up at the cost of steady state throughout. ..NET framework did a single compilation that attempted to balance start-up costs and steady state performance.

Tiered compilation allows the same method to have multiple compilations that can be swapped at runtime. One compilation can be aimed at fast start up while another is aimed at steady state. At start-up the JIT compiler generates a fast unoptimized compilation to facilitate fast start up. If the method is heavily used a background thread generates an optimised compilation that can be swapped in.

Most .NET core framework code loads from precompiled, ready to run images. These images lack some CPU optimisations. Where such methods are hot, they can also be recompiled at runtime for faster steady state performance.

On start-up time spent on JIT reduces by 35%. The amount of steady state performance probably depends how CPU bound the app is.

JSON Serialisation

Use Span and process UTF-8 directly without transcoding to UTF-16. For most tasks the JSON serializer is 1.5 to 3 times faster. System.Text.JSON.

Span<T>, Memory<T>

Span provides type-safe access to a contiguous area of memory. The memory can be located on the manager heap, the stack or even unmanaged memory. Span<T> is a ref struct which means it can never live on the manged heap. As such they cannot be boxed or assigned to variables of type object or interface types. They cannot be boxed or used as fields on classes or standard structs. The ref struct definition prevents any unnecessary heap allocations.

Span can be used to access substrings without allocation and copying.

```
string s = "John Smith";

// no allocation
System.ReadOnlySpan<char> span = "John Smith".AsSpan().Slice(5, 5);
// Allocation and copy
string sub = s.Substring(5, 5);
```

Internally a Span encapsulates a ref T that essentially is a direct pointer to some piece of memory. In this way it does not require an offset calculation to use it.

https://docs.microsoft.com/en-us/archive/msdn-magazine/2018/january/csharp-all-about-span-exploring-a-new-net-mainstay

https://channel9.msdn.com/Events/Connect/2017/T125

Parsing Integers

4x improvement in integer parsing

Queue Enqueue/Dequeue

Times 2 performance improvement over .net framework by removing expensive modulus operation

HTTP/2 Web Sockets

HTTP connection multiplexing. Concurrent requests across a single TCP connection.

Examples

Basics

HELLO WORLD

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;

namespace SingleTerminatingMiddleware
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var webHost = new WebHostBuilder()
                .UseKestrel()
                .Configure(ConfigureApp)
                .Build();

            // Start Listening
            webHost.Run();
        }

        private static void ConfigureApp(IApplicationBuilder bld)
        {
            bld.Run(TerminatingMiddleware);
        }

        private static async Task TerminatingMiddleware(HttpContext ctx)
        {
            await ctx.Response.WriteAsync("Hello World");
        }
    }
}
```

In bold is the terminating middleware that writes the response.

ADDING NON-TERMINATING MIDDLEWARE

We now add some non-terminating middleware to form a small pipeline

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;

namespace NonTerminatingMiddleware
{
    public class Program
    {
        public static void Main(string[] args)
        {
            IWebHost webHost = new WebHostBuilder()
                .UseKestrel()
                .Configure(ConfigureApp)
                .Build();

            webHost.Run();
        }

        private static void ConfigureApp(IApplicationBuilder applicationBuilder)
        {
             applicationBuilder.Use(NonTerminatingMiddleware);
            applicationBuilder.Run(TerminatingMiddleware);
        }

        private static async Task NonTerminatingMiddleware(HttpContext ctx,
            Func<Task> next)
        {
            // Preprocessing the request
            await next();
            // Pose-process the request
        }

        private static async Task TerminatingMiddleware(HttpContext httpContext)
        {
            await httpContext.Response.WriteAsync("Hello World");
        }
    }
}
```

STARTUP

```
public class Program
{
    public static void Main(string[] args)
    {
        var webHost = new WebHostBuilder()
            .UseKestrel()
            .UseStartup<Startup>()
            .Build();

        webHost.Run();
    }
}


public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    public void Configure(
            IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.Use(NonTerminatingMiddleware);
        app.Run(TerminatingMiddleware);
    }


    private static async Task NonTerminatingMiddleware(
        HttpContext ctx,
        Func<Task> next)
    {
        // Preprocessing the request
        await next();
        // Pose-process the request
    }

    private static async Task TerminatingMiddleware(
        HttpContext httpContext)
    {
        await httpContext.Response.WriteAsync("Using Startup.cs");
    }
}
```

ADDING OWN TERMINATING MIDDLEWARE

Program.cs

```
public class Program
{
    public static void Main(string[] args)
    {
        var webHost = new WebHostBuilder()
            .UseKestrel()
            .UseStartup<Startup>()
            .Build();

        webHost.Run();
    }
}
```

Startup.cs

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        app.Run(new MyTerminatingMiddleware().TerminatingMiddleware);
    }
}
```

MyTerminatingMiddleware.cs

```
public class MyTerminatingMiddleware
{
    public async Task TerminatingMiddleware(
        HttpContext ctx)
    {
        await ctx.Response.WriteAsync("My own terminating
middleware");
    }
}
```

ADDING OWN NON-TERMINATING MIDDLEWARE

Program.cs

```
public class Program
{
    public static void Main(string[] args)
    {
        var webHost = new WebHostBuilder()
            .UseKestrel()
            .UseStartup<Startup>()
            .Build();

        webHost.Run();
    }
}
```

Startup.cs

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        app.UseCustomMiddleware();
        app.Run(async context =>
        {
            await context
                .Response
                .WriteAsync("Custom Terminating Middleware");
        });
    }
}
```

CustomMiddleware

```
public class CustomMiddleware
{
    private readonly RequestDelegate _next;

    public CustomMiddleware(RequestDelegate next) => _next = next;

    public async Task Invoke(HttpContext ctx)
    {
        await _next(ctx);
    }
}
```

CustomMiddlewareExtensions.cs

```
public static class CustomMiddlewareExtensions
{
    public static IApplicationBuilder UseCustomMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<CustomMiddleware>();
    }
}
```

LOGGING

Security

Web API

A web API consists of a set of HTTP endpoints. Rather than return HTML they return something else which is typically JSON or XML. When building a Web API we take one of two approaches

- Remote Procedure Call
- REST (Representational State Transfer)

REST

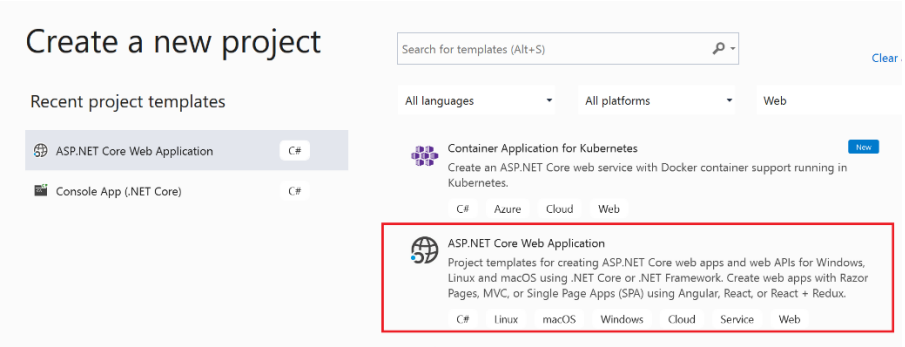
Representational State Transfer handles requests using HTTP verbs. The verbs act on resources specified as URL's.

- GET Get the specified resource
- POST Add a resource
- PUT Update the resource
- DELETE Delete the named resource

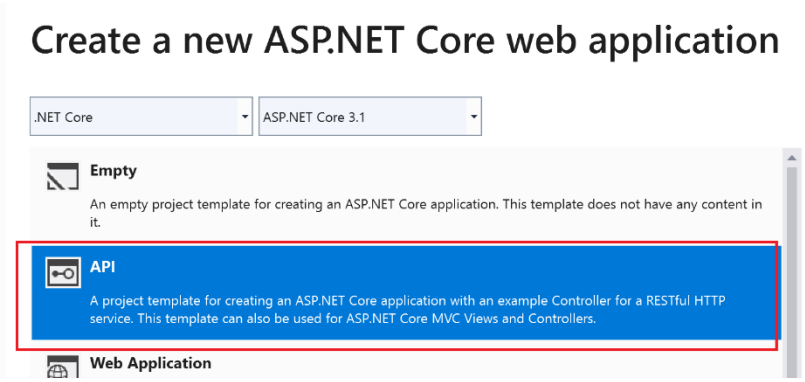
One interesting part of REST is that responses are expected to indicate whether or not they can be cached.

Creating Simple ASP.NET Rest API

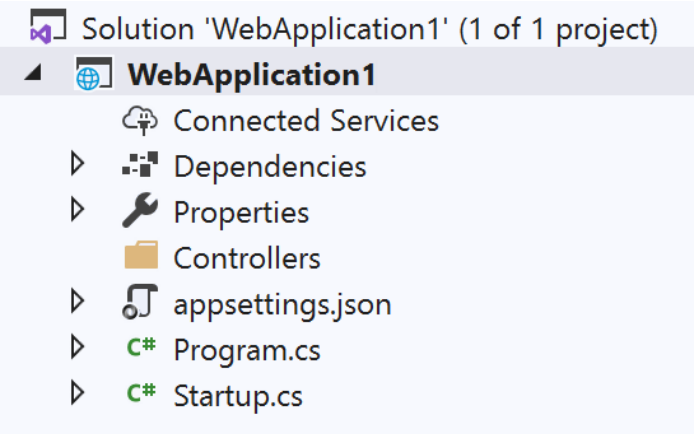
First open visual studio and create a new project as follows



Select a location and give it a name and hit create. Select the API template



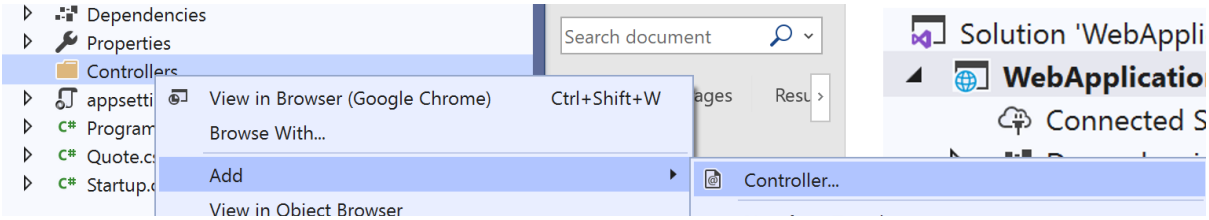
And hit the Create button. Delete any existing business objects and controllers. Your solution should look like this



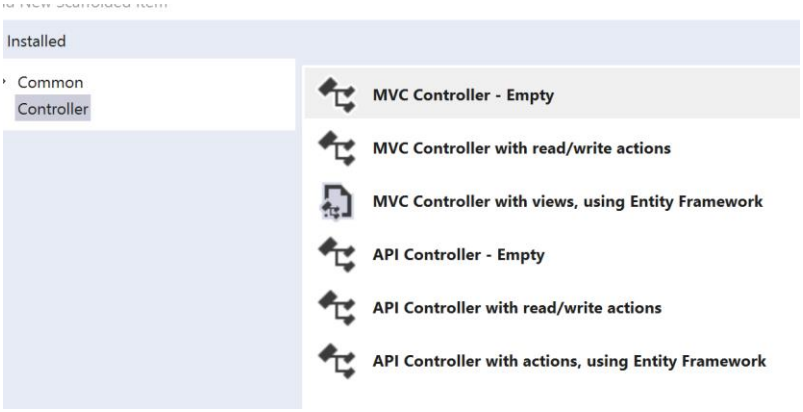
Now add an object called Quote.cs as follows.

```
namespace WebApplication1
{
    public class Quote
    {
        public int Id { get; set; }
        public double Price { get; set; }
    }
}
```

Right click on controllers and say Add controller



Now select API Contorller with read/write actions



Call it QuotesController.cs. Modify the code as follows

```
    [Route("api/[controller]")]
    [ApiController]
    public class QuotesController : ControllerBase
    {
        private static IDictionary<int, Quote> _quotes = new Dictionary<int, Quote>()
        {
            {1, new Quote() {Id = 1, Price = 100.0}}
        };

        // GET: api/Quotes
        [HttpGet]
        public IEnumerable<Quote> Get()
        {
            return _quotes.Values;
        }

        // GET: api/Quotes/5
        [HttpGet("{id}", Name = "Get")]
        public Quote Get(int id)
        {
            return _quotes[id];
        }

        // POST: api/Quotes
        [HttpPost]
        public void Post([FromBody] Quote value)
        {
            _quotes[value.Id] = value;
        }

        // PUT: api/Quotes/5
        [HttpPut("{id}")]
        public void Put(int id, [FromBody] Quote value)
        {
            _quotes[id] = value;
        }

        // DELETE: api/ApiWithActions/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
            _quotes.Remove(id);
        }
    }
```

Update the Properties/launchsettings.json

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:50685",
      "sslPort": 44389
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "api/quotes",
```

```
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "WebApplication1": {
      "commandName": "Project",
      "launchBrowser": true,
      "launchUrl": "api/quotes",
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Now we can run the project and see our quote.

Questions – Classic Interview

What performance improvements are there in .NET core

Tiered compilation

Many APIs take advantage of Span to improve performance.

What is tiered compilation

JIT compilation requires compromises

Aggressive optimization gives great steady state performance but increases start up time

Simple compilation gives fast start up and compromises steady state

Tiered compilation enables same method to be compiled multiple times.

First rough and fast and if the method is hot is can be compiled more aggressively

Questions – Web API

What is the difference between SOAP and REST?

SOAP is about RPC and REST is about acting on objects

SOAP uses only GET and POST