Kenny R N Wilson

.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 between .NET core and .NET Framework is that .NET Core can be deployed side by side with an application.

CLI

Everything needed to build, test, and run .net core applications is provided via the .NET Core Command-line Interface CLI. On windows this executable lives in the following.

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

The following table shows some useful commands.

Task	Command
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`

Building and Running a .NET Core Application

SourceCode

Please note the difference between the version of the CLI used to build the application and the version of the runtime targeted by the application.

CLI Version

By default, commands use the latest installed version of the CLI when building applications. 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": {
      "version": "5.0"
    }
}
```

Target Runtime

The version of the runtime is specified by the `TargetFramework` element of the `.csproj` file.

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

Running The Application

From the directory that contains the .csproj file we can run the application. Here we use the `Properties/launchSettings.json` to specify environment variables.

```
{
  "profiles": {
    "CommandLineApp": {
      "commandName": "Project",
      "environmentVariables": {
        "EnvVar": "World"
      }
    }
  }
}
```

We pass in command line arguments directly.

```
dotnet run --launch-profile CommandLineApp HelloWorld
```

Command Line Applications

Simple Hosted Service Command Line

SourceCode

In .NET Core we can use an instance of `IHost` to 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 run from inside `Program.Main`. The `Host.CreateDefaultBuilder()` creates an instance of `IHostBuilder` configured with a sensible set of defaults that work in many scenarios. I have created a project that uses these defaults in SourceCode. Below is the `Program.cs` file that uses `CreateDefaultBuilder()` to configure a sensible set of defaults.

Listing 1 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

SourceCode

If we can manually recreate what `CreateDefaultBuilder` does, we will better understand the .NET `IHostBuilder` 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 3 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 sources 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 as environment variables. So, see how it all comes together consider the following setup.

Listing 2 Properties/launchsettings.json

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

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 3 launchsettings.json

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

Listing 4 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)

SourceCode

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 object's 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

SourceCode

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

SourceCode

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

Hosting Models

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.

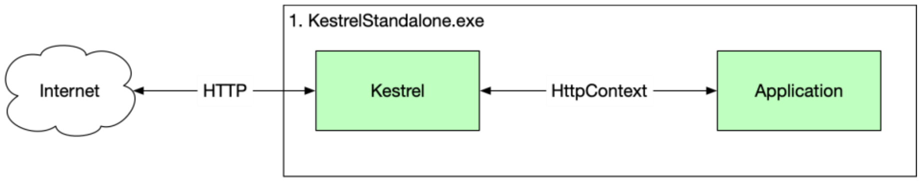
In addition to Kestrel, Windows also ships with two other in-process HTTP servers.

- 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

SourceCode



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 5 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.

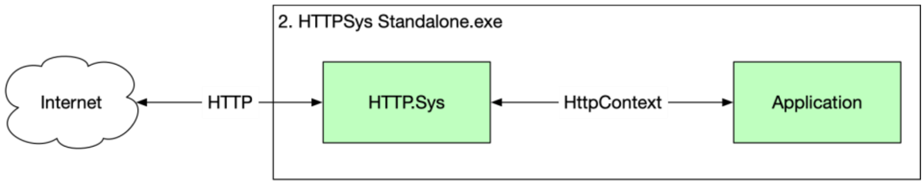
Listing 6 Properties/launchsettings.json

```
{
  "$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`

HTTP.SYS BY ITSELF

SourceCode



> **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 7 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.

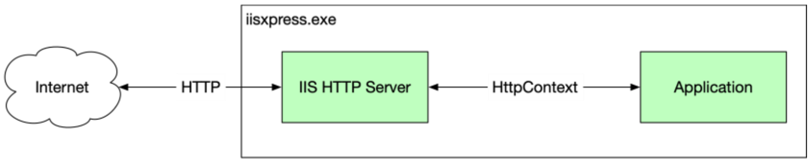
Listing 8 Program/launchSettings.json

```
{
  "$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`

IIS In process

SourceCode



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

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

Listing 9 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.

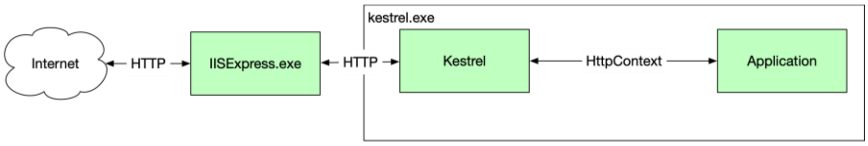
Listing 10 Properties/launchSettings.json

```
{
  "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 `IISEexpress.exe`

IIS OUT OF PROCESS

SourceCode



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

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

Listing 11 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 12 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`

Middleware

The order in which middleware handlers should be configured is shown in the following Microsoft documentation. We will only look at a subset of the middleware components as we generally are building APIs and not MVC type web sites.

Routing

SourceCode

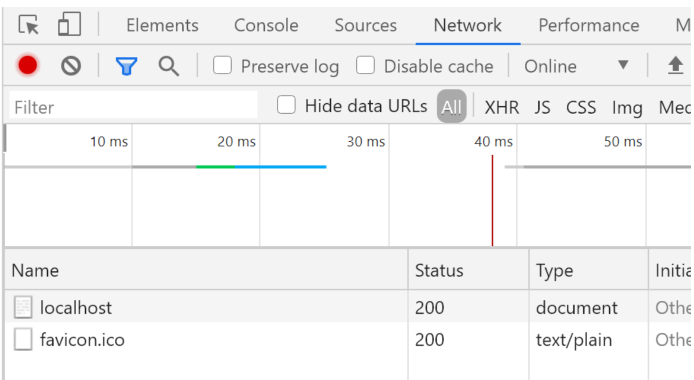
Cross Origin Resource Sharing (CORS)

SourceCode

Simple Authentication And Authorization

SourceCode

> **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.



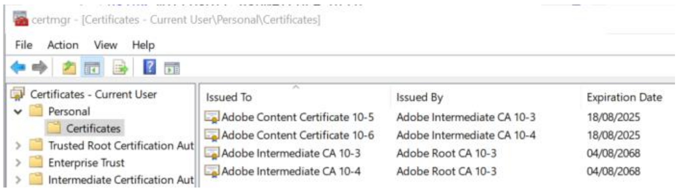
HTTPS And Certificates

HTTPS And Development Certificates

SourceCode

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`. The full

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

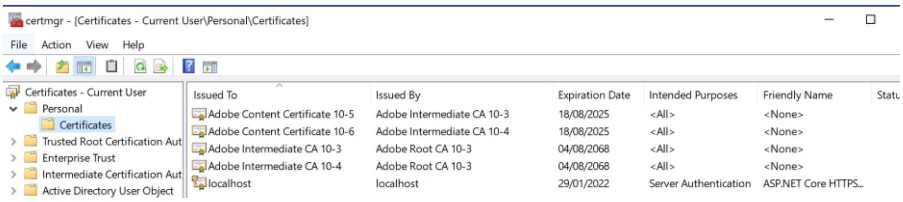


Now when I run the application, I see the following error telling me we have no certificate.

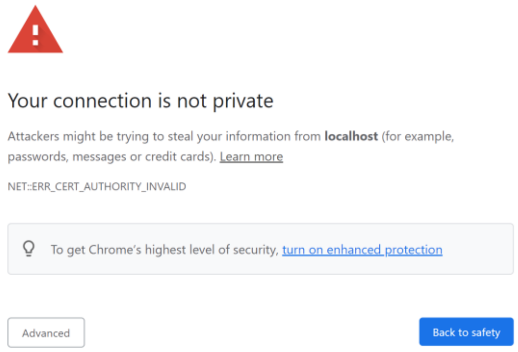
Let us go ahead and create a developer certificate as instructed.

```
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.



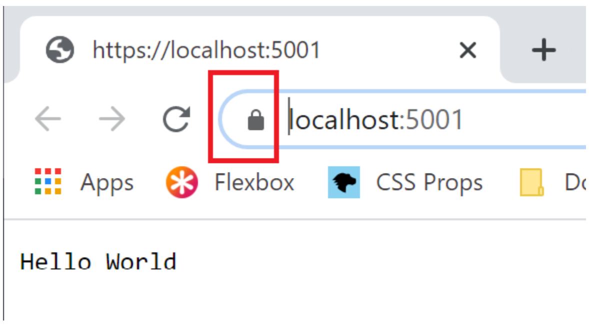
Now when we start out app from Visual Studio we see the following.



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

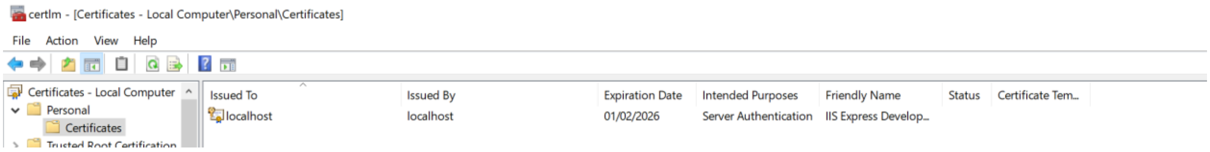
```
dotnet dev-certs https --trust
```

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

SourceCode

This simple project that highlights the NTLM protocol when used between a browser and our HTTP server. We have a controller with an authorized action. 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

SourceCode

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.

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

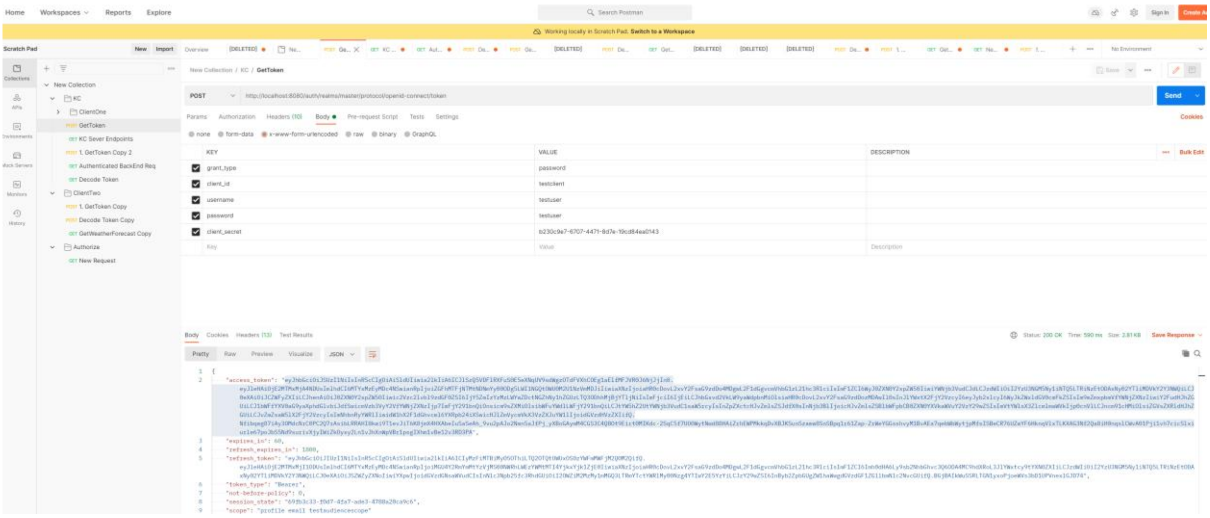
```
HTTP/1.1 200 OK
Date: Fri, 12 Feb 2021 08:47:25 GMT
Content-Type: text/plain; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked
WWW-Authenticate: Negotiate oRswGaADCgEAoxIEEAEAAADswe4CxIMi+gAAAAA=

Hello Kenny* Closing connection 0
```

KeyCloak / JWT

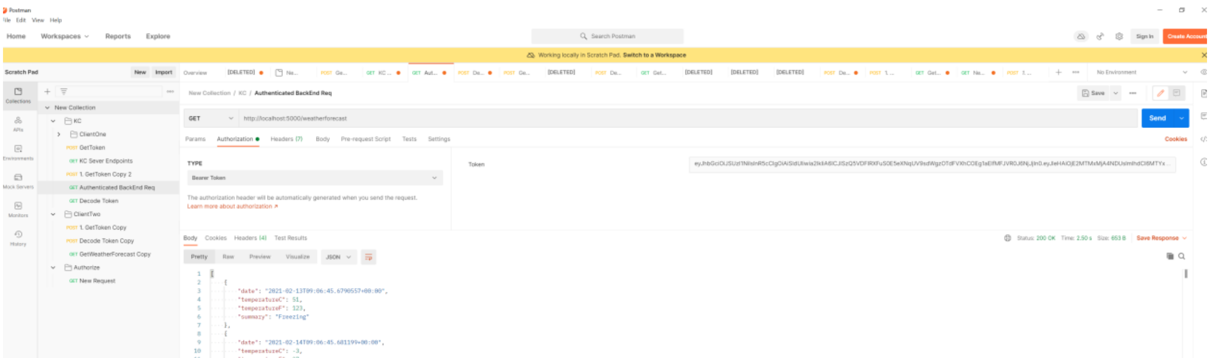
[SourceCode]

This code assumes we have setup a KeyCloak client as specified in my [KeyCloak notes].

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 and 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

[SourceCode]

If decorate our endpoint action with the Authorize attribute but give it no string argument it will use the default policy. The snippets in the following section are taken from a fully working project which is listed here.

Note how we do not specify a policy name argument to the Authorize attribute on the action.

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();
}

   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();
         });
   }
```

This means we will pick up the default policy configured in `Startup.cs` The other point to note is the authorization requirement we have is that the user is authenticated.

Custom Policy

In this example we show how to apply a custom policy and how some slightly more restrictive authorization than just requiring an authenticated use. The source code is here [Source Code]

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.