.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 HelloAuthorizationwith 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

List installed SDK versions	`dotnet --list-sdks`
List Installed Runtime versions	`dotnet --list-runtimes`

If one does not want to use the latest 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.

Secrets

To stop one having to add credentials to source files

https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets?view=aspnetcore-3.1&tabs=windows

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

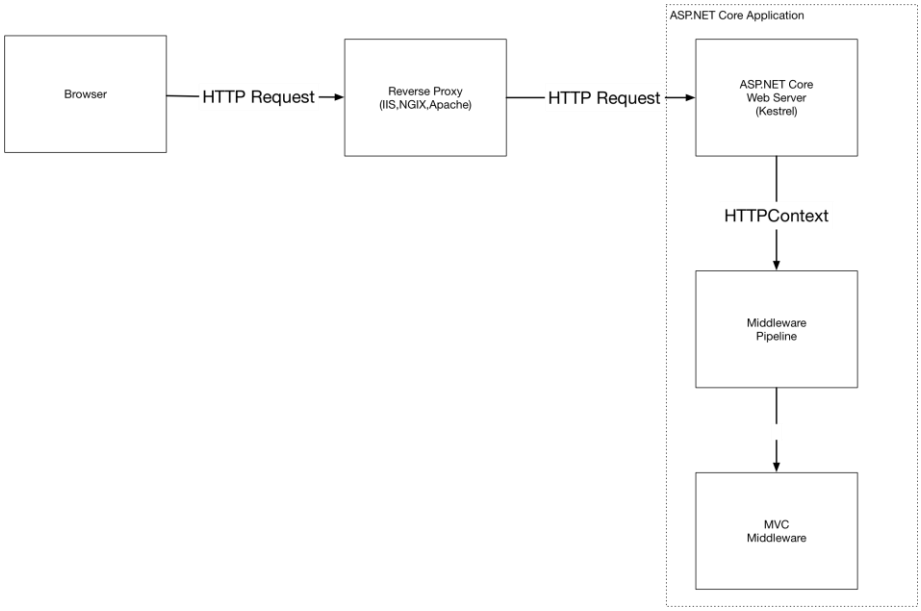
Times 2 performance improvement over .net framework by removing expensive modulus operation

HTTP/2 Web Sockets

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

ASP.NET Core

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



The simplest pipeline is one with 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(MiddlewareOne);
        applicationBuilder.Run(TerminatingMiddleware);
    }

    private static async Task MiddlewareOne(HttpContext httpContext,
        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");
    }
```

```
}
```

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.

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

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

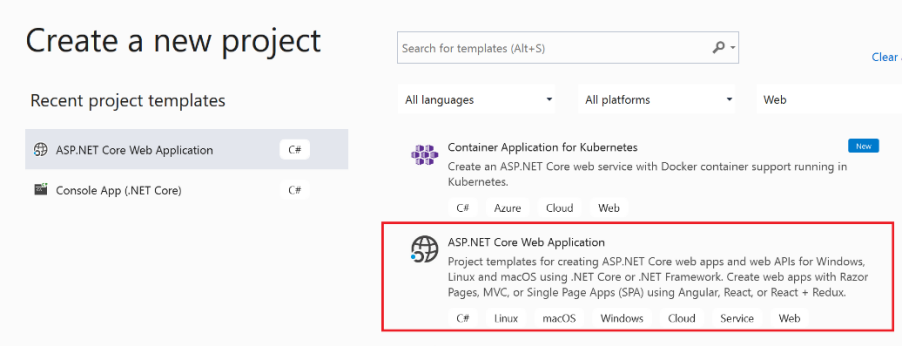
Representational State Transfer handles requests using HTTP verbs. The verbs act on resources specified as URL's.

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

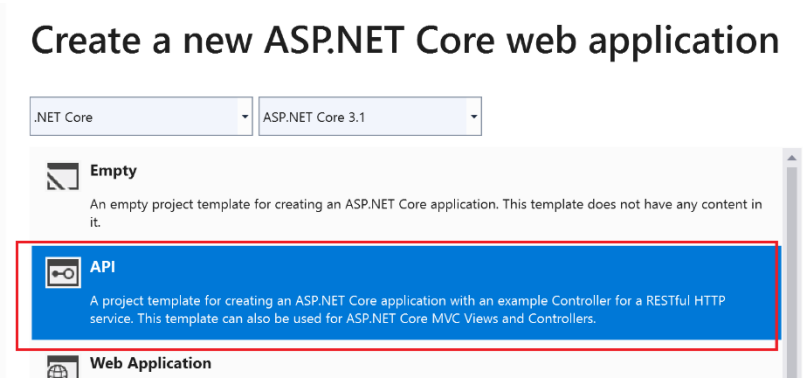
One interesting part of REST is that responses are expected to indicate whether or not they can be cached.

Creating Simple ASP.NET Rest API

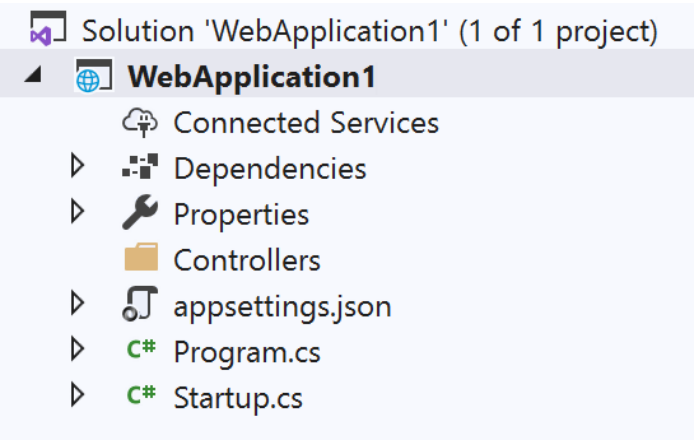
First open visual studio and create a new project as follows



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



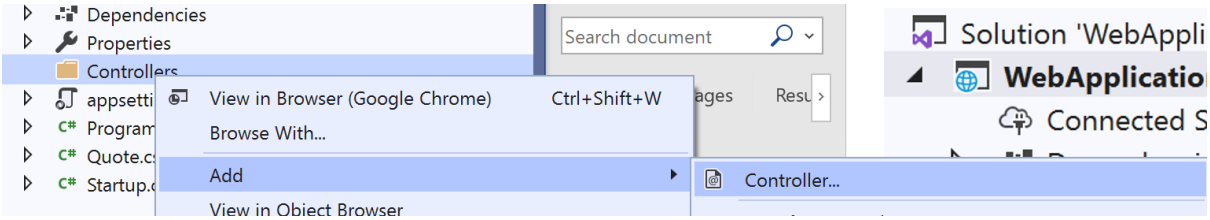
And hit the Create button. Delete any existing business objects and controllers. Your solution should look like this



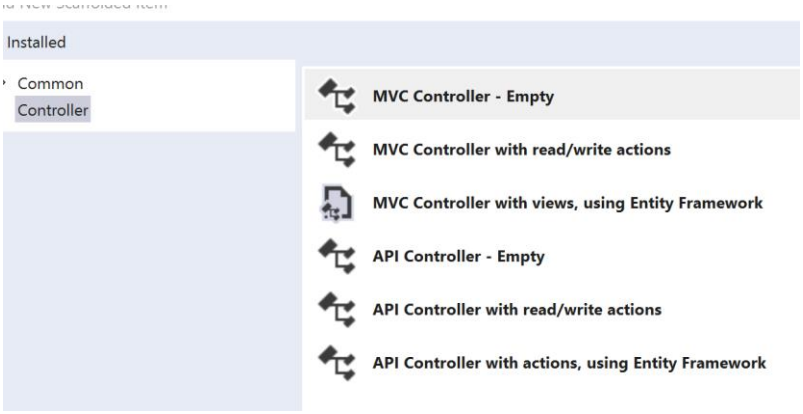
Now add an object called Quote.cs as follows.

```
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