**.NET Core**

**What is .NET Core?**

One of the main limitations of the .NET Framework was that it failed to share code across platforms. Now, however, .NET Core will provide developers with a library which can be deployed over various platforms and which also allows developers to import just the parts of the Framework they need for their projects.

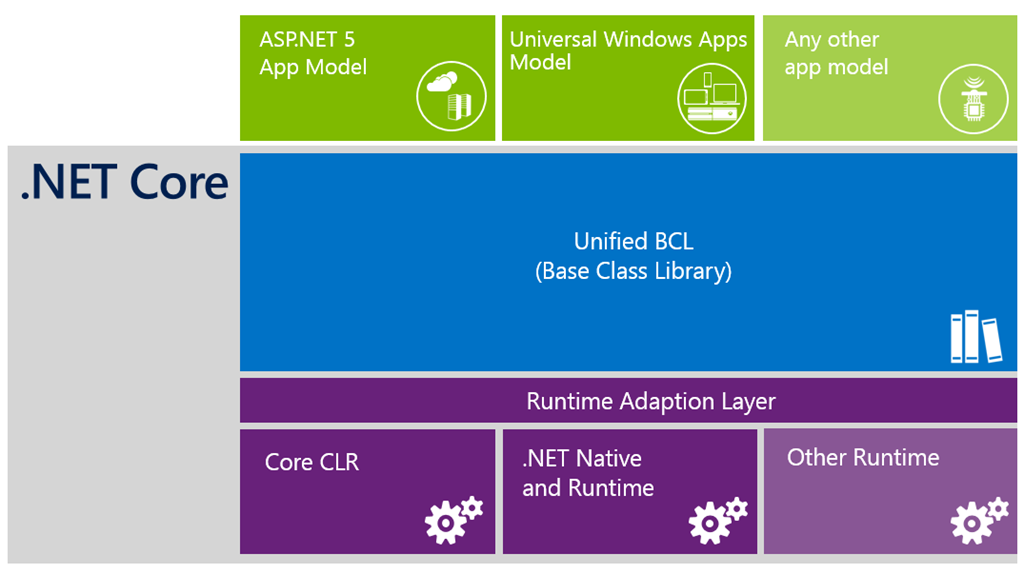
The .Net Core Runtime has been released to the open-source community who will collaborate with the .NET team to improve and extend the platform. Licensed under the MIT open source license, the code now provides developers with a fully supported, open source, cross platform .Net stack for creating server and cloud applications. This will not only include compilers and the CLR, but also the core .NET base class libraries and the higher level .NET Web, Data and API Frameworks.

**Characteristics of .NET Core**

.NET Core has the following characteristics:

* **Cross-platform**: Runs on Windows, macOS and Linux operating systems.
* **Consistent across architectures**: Runs your code with the same behavior on multiple architectures, including x64, x86, and ARM.
* **Command-line tools:** Includes easy-to-use command-line tools that be used for local development and in continuous-integration scenarios.
* **Flexible deployment**: Can be included in your app or installed side-by-side user- or machine-wide. Can be used with Docker containers.
* **Compatible**: .NET Core is compatible with .NET Framework, Xamarin and Mono, via .NET Standard.
* **Open source**: The .NET Core platform is open source, using MIT and Apache 2 licenses. .NET Core is a .NET Foundation project.
* **Supported by Microsoft**: .NET Core is supported by Microsoft, per .NET Core Support.

**Architecture**



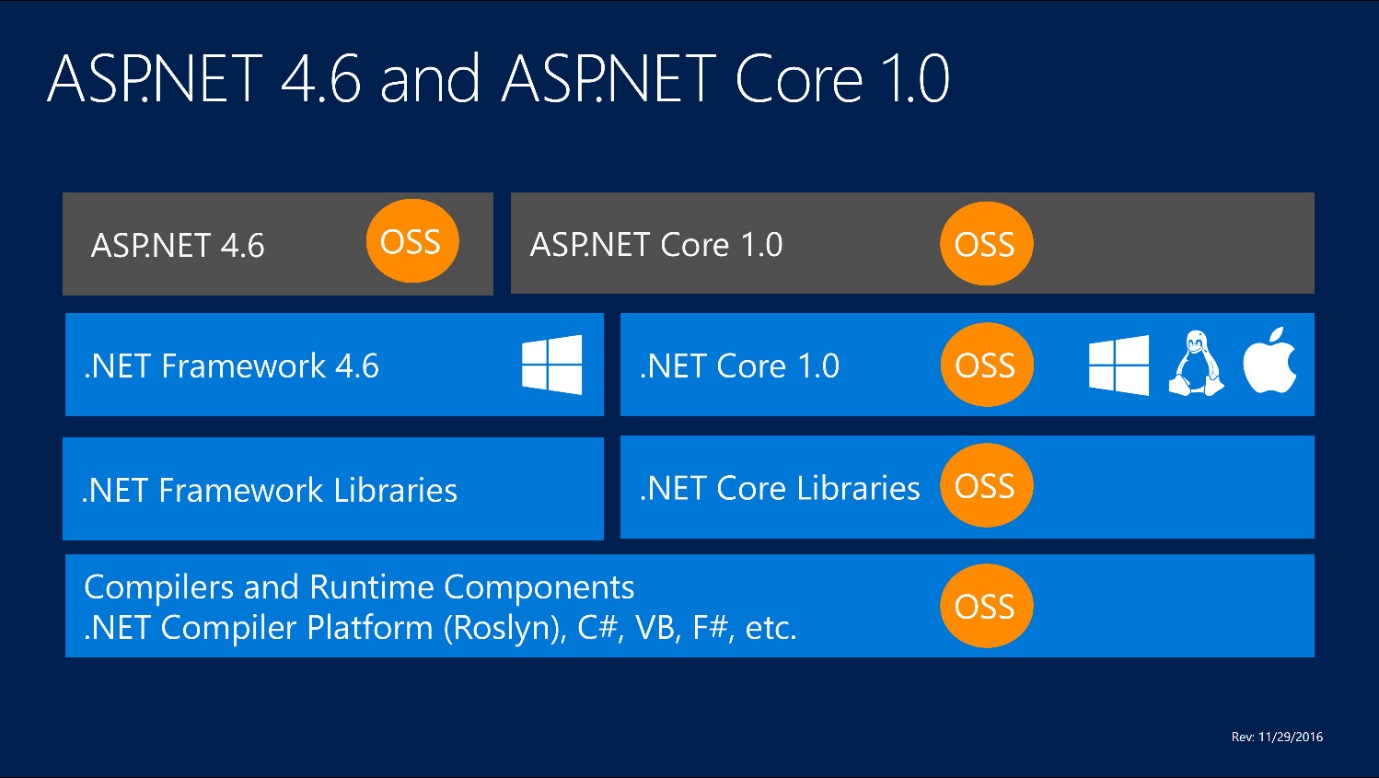
**.NET Core is composed of the following parts:**

* The **.NET Core runtime**, which provides a type system, assembly loading, a garbage collector, native interop and other basic services. .NET Core framework libraries provide primitive data types, app composition types and fundamental utilities.
* The **ASP.NET runtime**, which provides a framework for building modern cloud based internet connected applications, such as web apps, IoT apps and mobile backends.
* The **.NET Core CLI tools** and language compilers (Roslyn and F#) that enable the .NET Core developer experience.
* The **dotnet tool,** which is used to launch .NET Core apps and CLI tools. It selects the runtime and hosts the runtime, provides an assembly loading policy and launches apps and tools.

**When to use .NET Core?**

* Use .NET Core for your server application when:
* You have cross-platform needs.
* You are targeting microservices.
* You are using Docker containers.
* You need high-performance and scalable systems.
* You need side-by-side .NET versions per application.

**ASP.NET 4.6 vs ASP.NET Core**



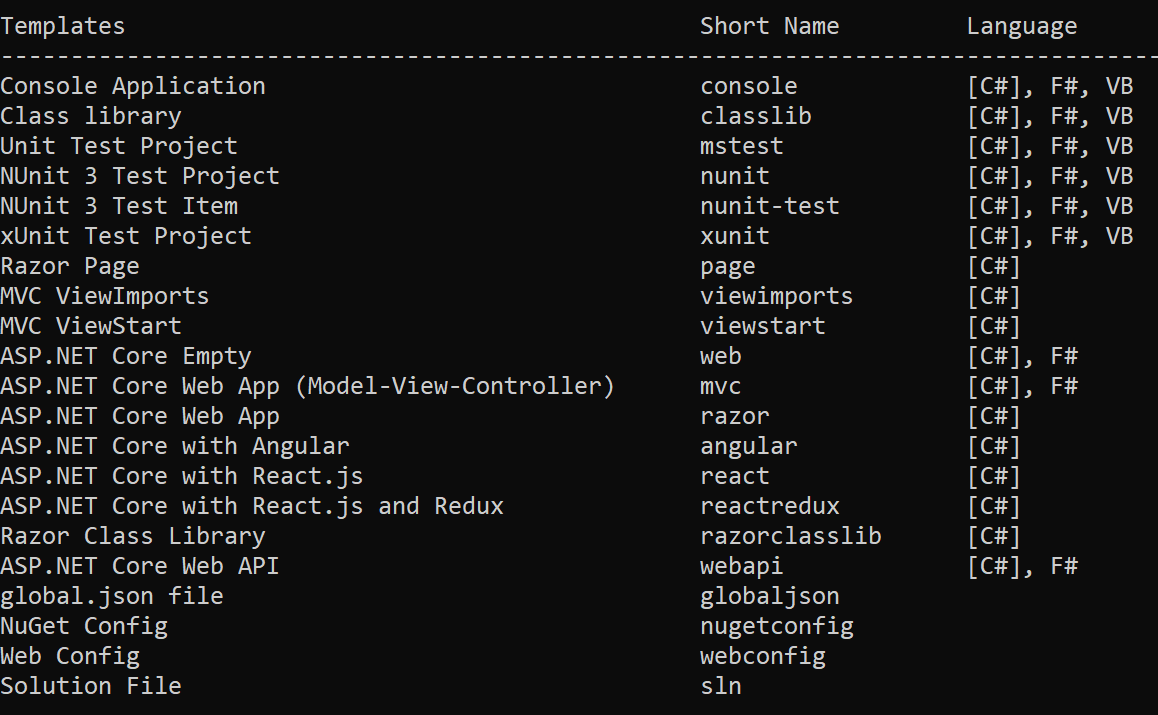
**.NET Core CLI Project**

Open the command prompt and type the following command. It will show the version and CLI informations.

**dotnet -v**

**Create new Project**

You can use the **dotnet new** command to create a project. It will display the types of project templates that you can create using dotnet CLI.



Run the following command to create a new console application

**dotnet new console -o <outputfolder>**

**Restore the packages**

**dotnet restore**

**Compile the code**

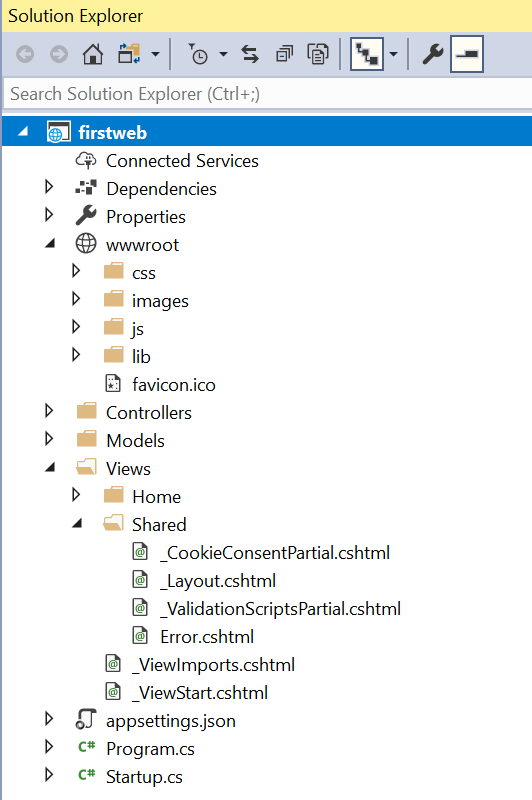
**dotnet build**

**Run the application**

**dotnet run**

**ASP.NET Core Web application**

Create a dotnet core web application by running the command ***dotnet new web***. This will create a new MVC project.



**Middleware**

Middleware is software that's assembled into an app pipeline to handle requests and responses. Each component:

* Chooses whether to pass the request to the next component in the pipeline.
* Can perform work before and after the next component in the pipeline is invoked.

Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.

Request delegates are configured using Run, Map, and Use extension methods. An individual request delegate can be specified in-line as an anonymous method (called in-line middleware), or it can be defined in a reusable class. These reusable classes and in-line anonymous methods are *middleware*, also called *middleware components*. Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the pipeline.



**Working with middleware**

Create a new Empty project in Visual Studio. In the Startup.cs file you can add the middleware in the Configure method.

**Run, Use and Map methods**

**Run**: Adds a terminal middleware delegate to the application's request pipeline.

Eg:

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World!<br/>");

});

**Use**: Adds a middleware delegate defined in-line to the application's request pipeline.

app.Use(async (context, next) =>

{

context.Response.Headers.Add("Content-Type",new StringValues("text/html"));

await context.Response.WriteAsync("First middleware<br/>");

await next.Invoke();

await context.Response.WriteAsync("Last middleware<br/>");

});

app.Use(async (context, next) =>

{

await context.Response.WriteAsync("Before processing<br/>");

await next.Invoke();

await context.Response.WriteAsync("After processing<br/>");

});

**Map**: Branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

Add the following method in the Startup class.

private static void SampleDelegate(IApplicationBuilder app)

{

app.Run(async context =>

{

await context.Response.WriteAsync("Hello from Sample<br/>");

});

}

Add the following code above the App.Run() method.

app.Map("/sample", SampleDelegate);

**Creating custom middleware class**

Add a new class to the project - ‘MyMiddleware’ class.

public class MyMiddleware

{

private readonly RequestDelegate \_next;

public MyMiddleware(RequestDelegate next)

{

\_next = next;

}

public async Task Invoke(HttpContext httpContext)

{

await httpContext.Response.WriteAsync("From custom middleware 'MyDelegate' <br/>");

await \_next.Invoke(httpContext);

}

}

Add a following line code above the app.Run() in the Configure() method of the Startup class. You can use the UseMiddleware<T>() method to configure the custom middleware.

app.UseMiddleware<MyMiddleware>();

**Configuring Custom middleware using Extension method**

To configure middleware using extension methods, create a new static class in the project and add the following method in it.

public static class CustomMiddlewareExtensions

{

public static IApplicationBuilder UseMyMiddleware(this IApplicationBuilder app)

{

return app.UseMiddleware<MyMiddleware>();

}

}

Now add the following line of code to the Configure method and remove the **app.UseMiddleware<MyMiddleware>()** statement.

app.UseMyMiddleware();

**Built-in middleware**

The following are some built-in middleware

1. Exception/error handling
2. HTTP Strict Transport Security Protocol
3. HTTPS redirection
4. Static file server
5. Cookie policy enforcement
6. Authentication
7. Session
8. MVC

**Serve static files**

Static files are stored within your project's web root directory. The default directory is <content\_root>/wwwroot, but it can be changed via the UseWebRoot method. Static files are accessible via a path relative to the web root. For example, the **Web Application** project template contains several folders within the *wwwroot* folder:

* **wwwroot**
  + **css**
  + **images**
  + **js**

Invoke the **UseStaticFiles** method within **Startup.Configure**:

public void Configure(IApplicationBuilder app)

{

app.UseStaticFiles();

}

**Serve files outside of web root**

Consider a directory hierarchy in which the static files to be served reside outside of the web root:

* **wwwroot**
  + **css**
  + **images**
  + **js**
* **MyStaticFiles**
  + **images**
    - *banner1.svg*

app.UseStaticFiles(new StaticFileOptions

{

FileProvider = new PhysicalFileProvider(

Path.Combine(Directory.GetCurrentDirectory(), "MyStaticFiles")),

RequestPath = "/StaticFiles"

});

**Enable directory browsing**

Directory browsing allows users of your web app to see a directory listing and files within a specified directory. Directory browsing is disabled by default for security reasons (see Considerations). Enable directory browsing by invoking the UseDirectoryBrowser method in Startup.Configure:

app.UseDirectoryBrowser(new DirectoryBrowserOptions

{

FileProvider = new PhysicalFileProvider(

Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),

RequestPath = "/MyImages"

});

Add required services by invoking the AddDirectoryBrowser method from Startup.ConfigureServices:

public void ConfigureServices(IServiceCollection services)

{

services.AddDirectoryBrowser();

}

**Serve a default document**

Setting a default home page provides visitors a logical starting point when visiting your site. To serve a default page without the user fully qualifying the URI, call the UseDefaultFiles method from Startup.Configure:

app.UseDefaultFiles();

UseDefaultFiles must be called before UseStaticFiles to serve the default file. UseDefaultFiles is a URL rewriter that doesn't actually serve the file. Enable the static file middleware via UseStaticFiles to serve the file.

The following code changes the default file name to mydefault.html:

public void Configure(IApplicationBuilder app)

{

// Serve my app-specific default file, if present.

DefaultFilesOptions options = new DefaultFilesOptions();

options.DefaultFileNames.Clear();

options.DefaultFileNames.Add("mydefault.html");

app.UseDefaultFiles(options);

app.UseStaticFiles();

}

**UseFileServer**

UseFileServer combines the functionality of UseStaticFiles, UseDefaultFiles, and UseDirectoryBrowser.

The following code enables the serving of static files and the default file. Directory browsing isn't enabled.

app.UseFileServer();

The following code builds upon the parameterless overload by enabling directory browsing:

app.UseFileServer(enableDirectoryBrowsing: true);

The following code enables static files, default files, and directory browsing of MyStaticFiles:

app.UseFileServer(new FileServerOptions

{

FileProvider = new PhysicalFileProvider(

Path.Combine(Directory.GetCurrentDirectory(), "MyStaticFiles")),

RequestPath = "/StaticFiles",

EnableDirectoryBrowsing = true

});

AddDirectoryBrowser must be called when the EnableDirectoryBrowsing property value is true:

public void ConfigureServices(IServiceCollection services)

{

services.AddDirectoryBrowser();

}

**Dependency Injection**

ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies.

Choose an appropriate lifetime for each registered service. ASP.NET Core services can be configured with the following lifetimes:

**Transient**

Transient lifetime services are created each time they're requested. This lifetime works best for lightweight, stateless services.

**Scoped**

Scoped lifetime services are created once per request.

**Singleton**

Singleton lifetime services are created the first time they're requested (or when ConfigureServices is run and an instance is specified with the service registration). Every subsequent request uses the same instance. If the app requires singleton behavior, allowing the service container to manage the service's lifetime is recommended. Don't implement the singleton design pattern and provide user code to manage the object's lifetime in the class.

**Third party DI engines in .NET Core**

ASP.NET Core ships with a simple built-in dependency injection container. Not only that, but the container itself is another dependency abstracted behind the **IServiceProvider** interface, easily allowing for authors of fully-featured containers to be compatible with the framework and for users to replace the built-in container.

**Default service container replacement**

The built-in service container is meant to serve the needs of the framework and most consumer apps. We recommend using the built-in container unless you need a specific feature that it doesn't support. Some of the features supported in 3rd party containers not found in the built-in container:

* Property injection
* Injection based on name
* Child containers
* Custom lifetime management
* Func<T> support for lazy initialization

**Configurations in .NET Core**

App configuration in ASP.NET Core is based on key-value pairs established by *configuration providers*. Configuration providers read configuration data into key-value pairs from a variety of configuration sources:

* Azure Key Vault
* Command-line arguments
* Custom providers (installed or created)
* Directory files
* Environment variables
* In-memory .NET objects
* Settings files

Adopt the following best practices:

* Never store passwords or other sensitive data in configuration provider code or in plain text configuration files.
* Don't use production secrets in development or test environments.
* Specify secrets outside of the project so that they can't be accidentally committed to a source code repository.

**Environments**

ASP.NET Core reads the environment variable ASPNETCORE\_ENVIRONMENT at app startup and stores the value in IHostingEnvironment.EnvironmentName. You can set ASPNETCORE\_ENVIRONMENT to any value, but three values are supported by the framework: Development, Staging, and Production. If ASPNETCORE\_ENVIRONMENT isn't set, it defaults to Production.

**Tag Helpers**

A tag helper is any class that implements the **ITagHelper** interface. However, when you author a tag helper, you generally derive from **TagHelper**, doing so gives you access to the Process method.