FREE eBook

LEARNING .net-core

#.net-core

Table of Contents

About

Chapter 1: Getting started with .net-core

Remarks

.NET Core is a general purpose development platform maintained by Microsoft and the .NET community on GitHub.

The following characteristics best define .NET Core:

- Flexible deployment: Can be included in your app or installed side-by-side user- or machine-wide.
- Cross-platform: Runs on Windows, macOS and Linux; can be ported to other OSes. The supported Operating Systems (OS), CPUs and application scenarios will grow over time, provided by Microsoft, other companies, and individuals. .NET can also be used in device, cloud, and embedded/IoT scenarios.
- Command-line tools: All product scenarios can be exercised at the command-line.
- Compatible: .NET Core is compatible with .NET Framework, Xamarin and Mono, via the .NET Standard Library.
- Open source: The .NET Core platform is open source, using MIT and Apache 2 licenses. Documentation is licensed under CC-BY. .NET Core is a .NET Foundation project.
- Supported by Microsoft: .NET Core is supported by Microsoft, per .NET Core Support

Composition

.NET Core is composed of the following parts:

- **A .NET runtime** which provides a type system, assembly loading, a garbage collector, native interop, and other basic services.
- **A set of framework libraries** which provide primitive data types, app composition types, and fundamental utilities.
- **A set of SDK tools and language compilers** that enable the base developer experience, available in the .NET Core SDK.
- **The 'dotnet' app host** which launches .NET Core apps. The app host selects and hosts the runtime, provides an assembly loading policy, and launches the app. The same host is also used to launch SDK tools in a similar fashion.

(Source: official documentation.)

Versions

.NET Core

Version	Release Date
1.0	2016-06-27
1.1.1	2017-03-07

Examples

Installation or Setup

Install .NET Core on macOS 10.11+, after install homebrew:

```
brew update
brew install openssl
mkdir -p /usr/local/lib
ln -s /usr/local/opt/openssl/lib/libcrypto.1.0.0.dylib /usr/local/lib/
ln -s /usr/local/opt/openssl/lib/libssl.1.0.0.dylib /usr/local/lib/
```

Install .NET Core SDK from https://go.microsoft.com/fwlink/?LinkID=835011

Official Microsoft .NET Core page with installation guides for Windows, Linux, Mac and Docker

Detailed instructions on getting .net-core set up or installed.

Building a Hello World Sample Application

Create an empty directory somewhere ...

```
mkdir HelloWorld
cd HelloWorld
```

Then use the built in scaffolding technology to create a Hello World sample

```
dotnet new console -o
```

This command creates two files:

- `HelloWorld.csproj` describes the project dependencies, settings, and Target Framework
- `Program.cs` which defines the source code for the main entry point and the console emitting of "Hello World".

If the `dotnet new` command fails, make sure you have installed .NET Core properly. Open the `Program.cs` file in your favorite editor to inspect it:

```
namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

To restore the project dependencies and the .NET runtime, execute

```
dotnet restore
```

To compile the application and execute it, enter

```
dotnet run
```

This last command prints "Hello World" to the console.

Installation from a binary archive

Note: These instructions are targeted at .NET Core 1.0.4 & 1.1.1 SDK 1.0.1 and higher.

When using binary archives to install, we recommend the contents be extracted to /opt/dotnet and a symbolic link created for dotnet. If an earlier release of .NET Core is already installed, the directory and symbolic link may already

```
sudo mkdir -p /opt/dotnet
sudo tar zxf [tar.gz filename] -C /opt/dotnet
sudo ln -s /opt/dotnet/dotnet /usr/local/bin
```

Ubuntu installation

```
dotnet-host-ubuntu-x64.deb
dotnet-hostfxr-ubuntu-x64.deb
dotnet-sharedframework-ubuntu-x64.deb
dotnet-sdk-ubuntu-x64.1.0.1.deb
```

Set up package source

The first step is to establish the source feed for the package manager. This is only needed if you have not previously set up the source or if you are installing on Ubuntu 16.10 for the first time.

Ubuntu 14.04 and Linux Mint 17

Commands

```
sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficmanager.net/repos/dotnet-release/
trusty main" > /etc/apt/sources.list.d/dotnetdev.list'
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 417A0893
sudo apt-get update
sudo apt-get install dotnet-dev-1.0.1
```

Installed packages

```
dotnet-host-ubuntu-x64.1.0.1.deb
dotnet-hostfxr-ubuntu-x64.1.0.1.deb
dotnet-sharedframework-ubuntu-x64.1.1.1.deb
dotnet-sdk-ubuntu-x64.1.0.1.deb
```

Ubuntu 16.04 and Linux Mint 18

Commands

```
sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficmanager.net/repos/dotnet-release/
xenial main" > /etc/apt/sources.list.d/dotnetdev.list'
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 417A0893
sudo apt-get update
sudo apt-get install dotnet-dev-1.0.1
```

Installed packages

```
dotnet-host-ubuntu.16.04-x64.1.0.1.deb
dotnet-hostfxr-ubuntu.16.04-x64.1.0.1.deb
dotnet-sharedframework-ubuntu.16.04-x64.1.1.1.deb
dotnet-sdk-ubuntu.16.04-x64.1.0.1.deb
```

Ubuntu 16.10

Commands

```
sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficmanager.net/repos/dotnet-release/
yakkety main" > /etc/apt/sources.list.d/dotnetdev.list'
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 417A0893
sudo apt-get update
sudo apt-get install dotnet-dev-1.0.1
```

Installed packages

```
dotnet-hostfxr-ubuntu.16.10-x64.1.0.1.deb
dotnet-host-ubuntu.16.10-x64.1.0.1.deb
dotnet-sharedframework-ubuntu.16.10-x64.1.1.1.deb
dotnet-sdk-ubuntu.16.10-x64.1.0.1.deb
```

source Official Documentation

Read Getting started with .net-core online: https://riptutorial.com/dot-net-core/topic/1024/getting-started-with--net-core

Chapter 2: .NET Core command line interface

Examples

Create .NET Core "Hello World" console project

Create a new **project.json** and example **Program.cs**:

```
dotnet new
```

Restore needed packages:

```
dotnet restore
```

Compile and run the example:

```
dotnet run
```

Publish and run a .NET Core project

Go to the **project.json** directory and publish:

```
dotnet publish
```

It will print the output directory of the operation, enter the directory and run the published project:

```
dotnet <project output>.dll
```

The default folder will be: `<project root>/bin/<configuration>/<target framework>/publish`

For example: `example/bin/Debug/netcoreapp1.0/publish`

If you have built the project previously, you can publish using:

```
dotnet --no-build publish
```

Important: Make sure you publish the project from the same user who restored the packages or you might publish it without the required libraries.

You can specify the configuration with the `-c <Configuration>` option. To publish in Release mode, use `dotnet publish -c Release`.

Scaffolding other project types

Using `dotnet new` will scaffold a new console application. To scaffold other types of projects, use the `-t` or `--type` flag:

```
dotnet new -t web
dotnet restore
dotnet run
```

The available templates vary by language.

C# Templates

- `console` (default) - A console application.
- `web` - An ASP.NET Core application.
- `lib` - A class library.
- `xunittest` - An xUnit test project.

F# Templates

- `console` (default) - A console application.
- `lib` - A class library.

Scaffolding projects in other languages

By default, `dotnet new` creates C# projects. You can use the `-l` or `--lang` flag to scaffold projects in other languages:

```
dotnet new -l f#
dotnet restore
dotnet run
```

Currently, `dotnet new` supports C# and F#.

Creating a NuGet package

To create a NuGet package from a project, run this command from a directory that contains **project.json**:

```
dotnet pack
```

The resulting `.nupkg` file will be named and versioned according to the properties in **project.json**. If there are multiple frameworks targeted in the project file, the package will support all of them.

Running automated tests

Running `dotnet test` from inside a folder that contains a test project will launch the test runner. The test runner will discover and run the tests in the project.

To be compatible with `dotnet test`, the **project.json** file must contain a `testRunner` property and a dependency on a compatible test runner package:

```
{
  "dependencies": {
    "dotnet-test-xunit": "2.2.0-preview2-build1029",
    "Microsoft.NETCore.App": {
      "type": "platform",
      "version": "1.0.0"
    },
    "xunit": "2.1.0"
  },
  "frameworks": {
    "netcoreapp1.0": {
      "imports": [ "dotnet", "portable-net45+win8" ]
    }
  },
  "testRunner": "xunit"
}
```

Read .NET Core command line interface online: https://riptutorial.com/dot-net-core/topic/4477/-net-core-command-line-interface

Chapter 3: .NET Core with Docker

Remarks

Fill with examples of using Docker on .NET Core platform, official base images for .NET Core application and self-hosted .NET Core app as well

Examples

Dockerfile sample

.NET Core app should be published using `dotnet publish`

```
FROM microsoft/dotnet:latest
COPY bin/Debug/netcoreapp1.0/publish/ /root/
EXPOSE 5000
ENTRYPOINT dotnet /root/sampleapp.dll
```

Read .NET Core with Docker online: https://riptutorial.com/dot-net-core/topic/4224/-net-core-with-docker

Chapter 4: Building libraries with .NET Core

Examples

Targeting .NET Standard

```
{
   "description": "My awesome library",
   "dependencies": {
      "NETStandard.Library": "1.6.0"
   },
   "frameworks": {
      "netstandard1.3": { }
   }
}
```

A library that targets `netstandard1.3` can be used on any framework that supports .NET Standard 1.3 **or later**. Choosing a lower .NET Standard version for a library means that more projects can use it, but less APIs are available.

Targeting both .NET Standard and .NET Framework 4.x

```
{
   "description": "My awesome library",
   "dependencies": { },
   "frameworks": {
      "net40": { },
      "netstandard1.3": {
         "dependencies": {
            "NETStandard.Library": "1.6.0"
         }
      }
   }
}
```

By targeting both `net40` and `netstandard1.3`, the library will work in both .NET 4.0+ projects and .NET Standard projects. It's important to move the `NETStandard.Library` dependency into the `netstandard1.3` section so it's only referenced when building for that framework.

Producing a NuGet package for a library

Any project that targets `netstandard1.x` can be packed into a NuGet package by running:

```
dotnet pack
```

The resulting package can be uploaded to NuGet, MyGet, or hosted in a local package source.

Platform-specific dependencies

You can specify different dependencies for each platforms:

```
"net45": {
    "frameworkAssemblies": {
        "System.Linq": "4.1.0"
    }
},
"netstandard1.3": {
    "dependencies": {
        "NETStandard.Library": "1.6.0",
        "System.Linq": "4.1.0-rc2"
    }
},
"netstandard1.4": {
    "dependencies": {
        "NETStandard.Library": "1.6.0",
        "System.Linq": "4.1.0"
    }
}
```

When this project is compiled and packed, each framework target will use a different set of dependencies:

- `net45` (projects targeting .NET 4.5+) will use the `System.Linq` assembly from the GAC.
- `netstandard1.3` (.NET Core projects targeting .NET Standard 1.3) will use the `NETStandard.Library` version 1.6.0 NuGet package, and the `System.Linq` prerelease version 4.1.0-rc2 NuGet package.
- `netstandard1.4` will use the same version of `NETStandard.Library`, but the release 4.1.0 version of `System.Linq`.

Adding a project reference to a library

If you have multiple libraries in the same solution, you can add local (project) references between them:

```
{
   "dependencies": {
      "NETStandard.Library": "1.6.0",
      "MyOtherLibrary": {
        "target": "project"
      }
   },
   "frameworks": {
      "netstandard1.3": { }
   }
}
```

The `target: project` property value tells NuGet to look in the current solution for `MyOtherLibrary`, instead of in your package sources.

Read Building libraries with .NET Core online: https://riptutorial.com/dot-net-core/topic/3290/building-libraries-with--net-core

Chapter 5: Components and Versioning in .NET Core

Introduction

This document covers the different components that make up a .NET Core distribution and how they are versioned. This document currently covers the 1.x releases.

Remarks

How components in .NET Core are versioned.

Components

.NET Core consists of multiple components that are each versioned independently and can often be mixed and matched.

- **Shared Framework**. This contains the APIs and the Virtual Machine and other runtime services needed for running .NET Core applications.
  - The current .NET Core Virtual Machine is called **CoreCLR**. This executes the .NET bytecode by compiling it JIT and provides various runtime services including a garbage collector. The complete source code for CoreCLR is available at https://github.com/dotnet/coreclr.
  - The .NET Core standard APIs are implemented in **CoreFX**. This provides implementations of all your favourite APIs such as `System.Runtime`, `System.Theading` and so on. The source code for CoreFX is available at https://github.com/dotnet/corefx.
- **Host** is also called the *muxer* or *driver*. This components represents the `dotnet` command and is responsible for deciding what happens next. The source for this is available at https://github.com/dotnet/core-setup.
- **SDK** is also sometimes called the *CLI*. It consists of the various tools (`dotnet` subcommands) and their implementations that deal with building code. This includes handling the restoring of dependencies, compiling code, building binaries, producing packages and publishing standalone or framework dependent packages. The SDK itself consists of the *CLI*, which handles command line operations (at https://github.com/dotnet/cli) and various subprojects that implement the various operations the CLI needs to do.

Components in .NET Core installations

Various official and unoffical packages, tarballs, zips and installers for .NET Core (including those available on https://dot.net/core) provide .NET Core in many variants. Two common ones are SDKs and Runtimes.

Each *SDK install* or *Runtime install* contains a number (possibly 0) of hosts, sdk and shared framework components described above.

- .NET Core **Runtime** contains
  - 1 version of *Shared Framework*
  - 1 version of the *Host*
- .NET Core **SDK** contains
  - 1 or more versions of the *Shared Framework* (varies depending on the version of the *SDK*)
  - 1 version of the *Host*
  - 1 version of the *SDK*

Examples

Identifying Versions

Each .NET Core component (SDK, Host and Shared Framework) is versioned independently.

You can find the version for each of them separately.

- **SDK**

  You can use the `--version` option to `dotnet` to see the SDK version. For example:

  ```
  $ ~/dotnet-1.1.1/dotnet --version
  1.0.0-preview2-1-003176
  ```

  `dotnet --info` also shows the SDK version.

- **Host**

  You can run `dotnet` by itself without any arguments or options to see the version of the host.

  ```
  $ ~/dotnet-1.1.1/dotnet

  Microsoft .NET Core Shared Framework Host

    Version  : 1.1.0
    Build    : 362e48a95c86b40cd1f2ef3d08741f7fed897956

  Usage: dotnet [common-options] [[options] path-to-application]
  ...
  ```

- **Shared Framework**

There no command currently to display the avaialble shared frameworks. I use `ls /path/to/where/you/installed/dotnet/shared/Microsoft.NETCore.App` which relies on internal implementation details. For example:

```
$ ls ~/dotnet-1.1.1/shared/Microsoft.NETCore.App/
1.1.1
```

Selecting Versions

It's possible to have multiple .NET Core *SDK*s and *Runtimes* available on disk. You can select the versions for each separately.

To select the version of the SDK to use, use `global.json`.

To select the version of the shared framework to use, target the specified framwork in the `.csproj` file (or `project.json` if you are still using that).

Read Components and Versioning in .NET Core online: https://riptutorial.com/dot-net-core/topic/9592/components-and-versioning-in--net-core

Chapter 6: Getting started with appsetting.json

Remarks

If you need more info, you can go and see official microsoft documentation

Examples

Simple configuration

Add this text to appsettings.json

```
{
  "key1": "value1",
  "key2": 2,

  "subsectionKey": {
    "suboption1": "subvalue1"
  }
}
```

Now you can use this configuration in your app, in the way like this

```
public class Program
{
    static public IConfigurationRoot Configuration { get; set; }
    public static void Main(string[] args = null)
    {
        var builder = new ConfigurationBuilder()
             .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");
        Configuration = builder.Build();

        Console.WriteLine($"option1 = {Configuration["key1"]}");
        Console.WriteLine($"option2 = {Configuration["key2"]}");
        Console.WriteLine(
            $"option1 = {Configuration["subsectionKey:suboption1"]}");
    }
}
```

Using configuration object for settings

Create class like a class below

```
public class MyOptions
{
    public MyOptions()
    {
        // Set default value, if you need it.
        Key1 = "value1_from_ctor";
    }
    public string Key1 { get; set; }
    public int Key2 { get; set; }
}
```

Then you need to add this code to your Startup class

```
public class Startup
{
    // Some default code here

    public IConfigurationRoot Configuration { get; set; }

    public void ConfigureServices(IServiceCollection services)
    {
        // Adds services required for using options.
        services.AddOptions();

        // Register the IConfiguration instance which MyOptions binds against.
        services.Configure<MyOptions>(Configuration);

    }
}
```

Then you could use it in your controllers in a way like presented below

```
public class TestController : Controller
{
    private readonly MyOptions _optionsAccessor;

    public TestController (IOptions<MyOptions> optionsAccessor)
    {
        _optionsAccessor = optionsAccessor.Value;
    }

    public IActionResult Index()
    {
        var key1 = _optionsAccessor.Key1 ;
        var key2 = _optionsAccessor.Key1 ;
        return Content($"option1 = {key1}, option2 = {key2}");
    }
}
```

Read Getting started with appsetting.json online: https://riptutorial.com/dot-net-core/topic/9057/getting-started-with-appsetting-json

Chapter 7: Installing .NET Core on Linux

Examples

Generic installation for Linux distributions

If you have one of the supported Linux distributions, you can follow the steps on the .NET Core website: https://www.microsoft.com/net

If you have an unsupported distribution:

Download the **.NET Core SDK** from the links, picking the distribution closer to the used one.

https://www.microsoft.com/net/download

If you have support for **deb** packages, you can install **Ubuntu/Debian** packages.

If you have support for **yum** packages, you can install **Fedora** packages.

Make sure your system has at least:

```
llvm-3.7.1-r3
libunwind-1.1-r1
icu-57.1
lttng-ust-2.8.1
openssl-1.0.2h-r2
curl-7.49.0
```

Read Installing .NET Core on Linux online: https://riptutorial.com/dot-net-core/topic/3354/installing--net-core-on-linux

Chapter 8: The global.json

Remarks

The `global.json` file is extremely powerful and unique to `.NET Core` and `ASP.NET Core` applications.

Examples

Schema

Schema store:

```
{
  "title": "JSON schema for the ASP.NET global configuration files",
  "$schema": "http://json-schema.org/draft-04/schema#",

  "type": "object",
  "additionalProperties": true,
  "required": [ "projects" ],

  "properties": {
    "projects": {
      "type": "array",
      "description": "A list of project folders relative to this file.",
      "items": {
        "type": "string"
      }
    },
    "packages": {
      "type": "string",
      "description": "The location to store packages"
    },
    "sdk": {
      "type": "object",
      "description": "Specify information about the SDK.",
      "properties": {
        "version": {
          "type": "string",
          "description": "The version of the SDK to use."
        },
        "architecture": {
          "enum": [ "x64", "x86" ],
          "description": "Specify which processor architecture to target."
        },
        "runtime": {
          "enum": [ "clr", "coreclr" ],
          "description": "Chose which runtime to target."
        }
      }
    }
  }
}
```

Read The global.json online: https://riptutorial.com/dot-net-core/topic/1569/the-global-json

Chapter 9: Understanding System.Runtime vs. mscorlib

Remarks

Every .NET library and programming language utilize a set of elementary data types like `System.Int32`, `System.Object`, `System.Type` or `System.Uri`. These data types form the base of all other structures including all custom written .NET libraries. All these types are hosted in a base library, which is either `mscorlib` or `System.Runtime`.

The libraries which can be used with .NET Core are based on `System.Runtime` core library while for the .NET Framework (the Windows component) they are based on `mscorlib`. This essential difference lead to...

- the incompatibility of older libraries since they expect a type `System.Object, mscorlib` while a .NET Core library would expect `System.Object, System.Runtime`.
- a type forwarding facade library called `System.Runtime` to the `mscorlib` in the **.NET Framework**. This library is otherwise (nearly) empty but enables the usage of `System.Runtime` based PCL libraries on the .NET Framework.
- a type forwarding `mscorlib` to the `System.Runtime` in a **future** version of **.NET Core**.
- the introduction of the portable class library concept (PCL) and as a second generation the `netstandard` as a method of unification between the two core libraries.

AND out of that, countless questions on Stack Overflow.

Examples

Popular Error: Misleading NuGet the wrong way

.NET Core `project.json` supports NuGet importing (a.k.a. lying according to this SO answer). It is impossible to include a `mscorlib` based library due to an `import` statement.

```
{
    "version": "1.0.0-*",

    "dependencies": {
        "Microsoft.AspNet.Identity.EntityFramework": "2.2.1",
        "NETStandard.Library": "1.6.0"
    },

    "frameworks": {
        "netstandard1.6": {
            "imports": [
                "net461"
            ]
        }
    }
}
```

Imports only work with portable class libraries (which are `System.Runtime` based) or deprecated target framework monikers which are also `System.Runtime` based (e.g. `dotnet` or `dnxcore`)

Popular Error: Add a NuGet package which was not made for netstandard / netcoreapp (System.Runtime)

In the example `project.json` below, an assembly `Microsoft.AspNet.Identity.EntityFramework` was added which is `mscorlib` based.

```
{
    "version": "1.0.0-*",

    "dependencies": {
        "Microsoft.AspNet.Identity.EntityFramework": "2.2.1",
        "NETStandard.Library": "1.6.0"
    },

    "frameworks": {
        "netstandard1.6": { }
    }
}
```

The author of the assembly `Microsoft.AspNet.Identity.EntityFramework` has not ported the NuGet package yet to `netstandard` (they actually did it, they just renamed the package as well `Microsoft.AspNetCore.Identity.EntityFrameworkCore` ;))

If you encounter a package you need and is not yet on netstandard, please contact the author and - if possible - help porting it.

Popular Error: Misunderstanding the outcome

Targeting multiple frameworks with `project.json` is simple. However the result are two different compilations. Take the following example:

```
{
    "version": "1.0.0-*",

    "dependencies": {
        "NETStandard.Library": "1.6.0",
        "System.Collections.Immutable": "1.2.0"
    },

    "frameworks": {
        "netstandard1.3": { },
        "net451": { }
    }
}
```

The compilation process for the `project.json` file will lead to two resulting artifacts:

- One compiled dll for the `System.Runtime` based `netstandard` world which can be used on .NET Core, .NET Framework (via type forwarders) and Xamarin products (via type forwarders). This dll has references to `System.Runtime` and `System.Collections.Immutable`.
- Another compiled dll directly for the `mscorlib` based .NET Framework. This dll will have references to `mscorlib` and `System.Collection.Immutable`.

However, it is important to understand that the `netstandard1.0` based `System.Collections.Immutable` will utilize different `System.Runtime` implementations for each build dll at runtime. The `System.Runtime` which comes with .NET Core does not have any assembly dependencies on its own (since it implements the core library). The `System.Runtime` used for with the .NET Framework has references (for the type forwarders) to the .NET Framework assemblies `mscorlib`, `System.Core`, `System` and `System.ComponentModel.Composition`.

Popular Error: Accidently adding a mscorlib library as a dependencies to a netstandard/netcoreapp

Another popular error is the referring of packages which does not satisfy all framework on the global scope when multiple frameworks are targeted.

```
{
    "version": "1.0.0-*",

    "dependencies": {
        "NETStandard.Library": "1.6.0",
        "Microsoft.AspNet.Identity.EntityFramework": "2.2.1"
    },

    "frameworks": {
        "netstandard1.3": { },
        "net451": { }
    }
}
```

The (meta) library `NETStandard.Library` works fine in this example, since it targets both `netstandard1.3` and `net451`. However the library `Microsoft.AspNet.Identity.EntityFramework` does only target the .NET Framework `net` and `mscorlib` and therefore cannot be used for a `netstandard` output.

Either search for a library (version) which cover both frameworks or add the library in the conditional dependencies below the framework.

```
{
    "version": "1.0.0-*",

    "dependencies": {
        "NETStandard.Library": "1.6.0"
    },

    "frameworks": {
        "netstandard1.3": { },
        "net451": {
            "dependencies": {
                "Microsoft.AspNet.Identity.EntityFramework": "2.2.1",
            }
        }
    }
}
```

In this case, the library can only be used in conditional #ifdef blocks for the `net451` build.

Read Understanding System.Runtime vs. mscorlib online: https://riptutorial.com/dot-net-core/topic/5994/understanding-system-runtime-vs--mscorlib

Credits

S. No	Chapters	Contributors
1	Getting started with .net-core	4444, alvaro, Community, Eidolon, gbellmann, hmnzr, jnovo, Max Cheetham, Ming Tong, Thomas
2	.NET Core command line interface	Nate Barbettini, Stefano d'Antonio
3	.NET Core with Docker	hmnzr
4	Building libraries with .NET Core	Chris Forbes, hmnzr, Nate Barbettini
5	Components and Versioning in .NET Core	omajid
6	Getting started with appsetting.json	Egorikas
7	Installing .NET Core on Linux	Stefano d'Antonio
8	The global.json	David Pine
9	Understanding System.Runtime vs. mscorlib	David Pine, Thomas