.NET Core Succinctly®

by Giancarlo Lelli

Syncfusion | Technology Resource Portal

.NET Core Succinctly

By

Giancarlo Lelli

Foreword by Daniel Jebaraj

Technical Reviewer: Gavin Lanata
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Tres Watkins, content development manager, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the author

Giancarlo Lelli is addicted to Microsoft technologies. He writes code for a living, but when he's not writing code for work, he writes code to relax and learn something new. He enjoys movies, rock music, and running outdoors.

He graduated in Computer Science at the University of Cagliari, where he fulfilled the role of Microsoft Student Partner, and at a later stage, Xamarin Student Ambassador.

Giancarlo writes technical articles and shares technical tips for the ASPItalia Network, and back in 2012 he founded, and now leads, a Microsoft community based in Cagliari called the Italian Developer Connection. Along with the other co-founders, he organizes free technical events about Microsoft technologies, and as a Community Lead, helps the Italian branch of Microsoft to engage with local developers, students, and independent experts. Giancarlo took part as speaker in many nationwide technical events, such as Community Days, Startup Revolutionary Road, Microsoft DevCamps, Future Decoded, and the Windows Professional Conference from 2014 to 2015.

Giancarlo currently works for Avanade Italy in Italy's Delivery Center. As an Analyst, his job consists of building enterprise-grade solutions based on the Microsoft Dynamics CRM platform.

He's been a Microsoft MVP on the Windows Platform since 2015. He rumbles about technology and his life on Twitter @itsonlyGianca.

Preface

Before digging deeper into the contents of this e-book, I believe that we should first analyze why a book like this was considered useful and needed at this point in time. At first, this might sound like a pretentious goal, but during the course of this e-book, and for the most part in this introductory chapter, we will understand that at the end of the day it was quite clear and somehow obvious.

I believe that the purpose of technology is to help individuals achieve their goals, whether they use it consciously or not. My job, and probably yours, since you're reading this, is about leveraging technology to build stuff that at the end of the day will empower someone. We are witnesses of this momentum on a daily basis with the unstoppable growth of mobile technologies, platforms, and devices. Devices are announced and their computational power increases every year (if not every six months) and we, first as consumers and then as developers, are faced with the never-ending challenge of choosing what to build, how to build it, and for whom. I believe that the most sensitive aspect of this is that any of those three choices inevitably comes with a price…and many times we pay with our most precious currency, time. Since time is a finite resource and there's no way to earn time, we must be sure not to waste it. Rephrasing this is a more technological way, we must value our previous investments in the best way we can.

However, until recent days, this hasn't been an easy task. Let's analyze what used to happen, and what we as Microsoft developers were used to facing. Twelve years since the release of the first .NET framework, we were used to having fragmented versions of .NET for different platforms. From the .NET Compact Framework to Silverlight, Windows Phone, and Windows Store applications, every time Microsoft has taken .NET to a new platform, the supposedly "common" language runtime has ended up with a different subset. Each time, there's a different runtime, framework, and application model, with different development being done at each layer and APIs that go back to a common code base but haven't always stayed common. Back in those days, Microsoft tried endlessly to reassure us, talking about common skill and reusing what we already knew, but the reality was one: fragmentation (and many headaches).

Microsoft tried to tackle this problem by introducing Portable Class Libraries, Shared Projects, and finally, Universal Apps. The core concept of the first two approaches was the idea of contract, or in other words, a way of abstracting the APIs so you can use them as if they were the same for each platform. These approaches ended up becoming standard de-facto in modern application developments, and once Xamarin came up, it allowed us to take this even further by letting us reuse our code on other platforms like iOS, Android, and Mac OS X. That was a sort of glimpse into a cross-platform .NET. Unfortunately, in each case, the implementation of those APIs was different. At this point you might think that the solution to our dilemma was close, but unfortunately it was not…or at least not a complete one. Portable Class Libraries managed to get platforms closer, but were just not capable of making the .NET Framework more modular.

Another critical point was backward compatibility between different versions of the framework, even in the same platform. Adding, let’s say, an interface or an overload to a method wasn’t just about writing a few lines of code and releasing a new version of the framework. It was more about being sure that the change wouldn’t cause any trouble to existing apps. Needless to say, this fear is more than enough to freeze the design of any new feature.

In order to figure out a way to overcome these limitations (modularity and backward compatibility), Microsoft started to ship some components of the .NET Framework as standalone packages on NuGet (Immutable Collection library), allowing people to use it and provide feedback. This ended up being a huge success; once the API reached the final version, it ended up having a better design. And thanks to NuGet backward compatibility, it was something that came out of the box, since NuGet allows you to install specific versions of each package.

However, how do all these considerations fit in the realm of this e-book, or more specifically, in the realm of .NET Core? Well, as we will see in the upcoming chapters .NET Core is the product of a new Microsoft, a Microsoft that embraces openness and puts us developers or customers on top of its priorities. So why did I believe that all these considerations would have led to an obvious answer? The answer is easy, and I believe it relies on what should be considered the definition of .NET Core: A cloud-optimized, cross-platform, and open source port of the .NET Framework…basically what an allegedly modern framework should look like in 2016.

Introduction

Thanks to the preface in the previous chapter, we may have come to realize why .NET Core was needed, and how .NET Core is positioned inside the bigger picture. Without further ado, we are now ready to talk about .NET Core.

The bigger picture

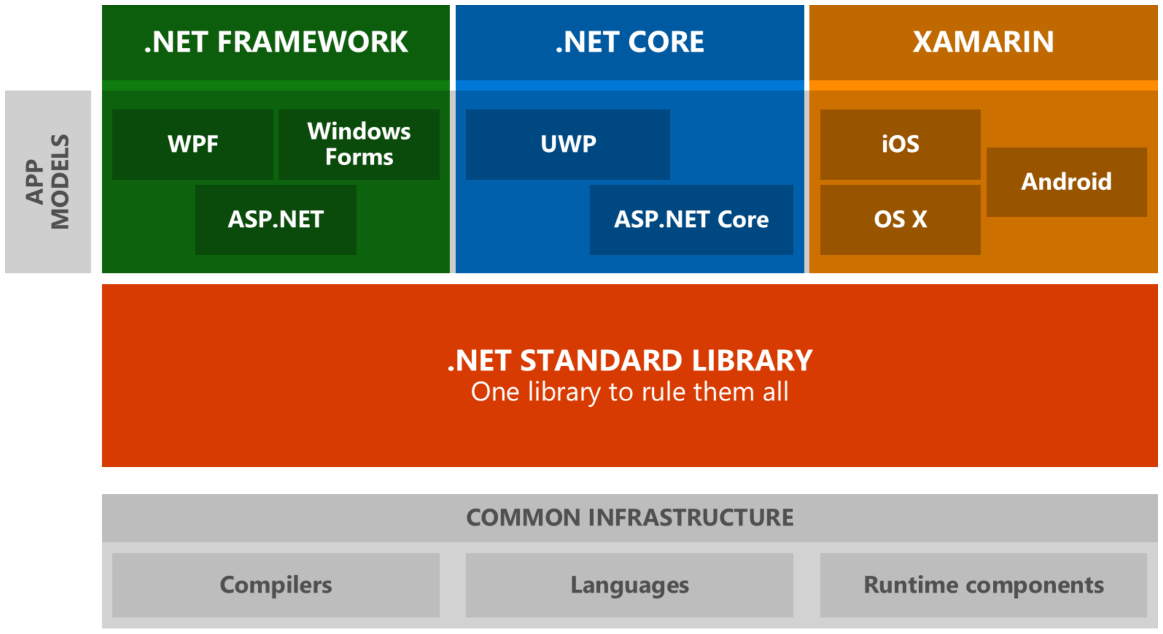


Figure 1: The 2016.NET Panorama

As you can see in Figure 1, the .NET panorama is rich in app models that are clustered inside a specific, let's say, technology stack. Not considering Xamarin (which by itself would require another book), we are left with two main clusters: the full .NET Framework and .NET Core.

They both sit on top the .NET Standard Library and share a common infrastructure that includes all the technologies that power the compiler, the languages, and the runtime components. ASP.NET, per se, is present in both clusters; however, there is a substantial difference between the two—the version, or maybe the type, of .NET Framework the runtime uses.

ASP.NET Core runs using the cross-platform version of the .NET Framework, which is the subject of this book, while ASP.NET uses the full .NET Framework. If you'd like to know more about this, StackOverflow is a good place to start. Let's now analyze the section that refers to the common infrastructure.

.NET Core Runtime (CoreCLR)

The runtime implementation of .NET Core is called CoreCLR. The CoreCLR is a unique term that groups together all the technologies that are fundamental to the runtime, such as RyuJIT, the .NET Garbage Collector, native interop, and many other components. The CoreCLR is cross-platform, with multiple OS and CPU ports in progress. Since it's open sourced, you can find the official repo here.

The Just-In-Time compiler: RyuJIT

We find ourselves in a place in time where we can have (virtually) unlimited computing power and resources. Not long ago, RAM was relatively expensive, and that was somehow fine since the x86 architecture was living its golden days. However, as time has passed, the prices of RAM have decreased, allowing the majority of computers (and in recent years, also smartphones and IoT boards) to adopt the x64 bit architecture. Thanks to its long addresses, a x64 architecture can index a probably infinite amount of RAM. The rapid growth of the x64 architecture was somehow an edge case. Needless to say, this tiny detail had a repercussion on the .NET Framework. In fact, the .NET x64 Jitter was originally designed to produce very efficient code throughout the long run of a server process, while the .NET x86 JIT was optimized to produce code quickly so that the program starts up fast. The line between client and server got a little blurred, and the current implementation of the JIT needed a little tweak.

But why do we need a JIT compiler? And why does it need to be efficient? We need a JIT compiler because before we can run any Microsoft intermediate language (MSIL) assembly, we must first compile it against the common language runtime to native code for the target machine architecture. The more efficient this compilation process is, the faster and optimized are our assemblies.

With this is mind, the .NET code generation team worked really hard, and eventually came up with a next-generation x64 compiler, codenamed RyuJIT.

RyuJIT is the next generation Just-In-Time (JIT) compiler for .NET. It uses a high-performance JIT architecture, focused on high throughput, JIT compilation. It's much faster than the existing JIT64 64-bit JIT that has been used for the last 10 years (introduced in the 2005 .NET 2.0 release). There was always a big gap in throughput between the 32- and 64-bit JITs. RyuJIT is similarly integrated into .NET Core as the 64-bit JIT. This new JIT is twice as fast, meaning apps compiled with RyuJIT start up to 30 percent faster. RyuJIT is based off of the same codebase as the x86 JIT, and in the future it will be the basis of all of Microsoft's JIT: x86, ARM, MDIL, and whatever else comes along. Having a single codebase means that .NET programs are more consistent between architectures, and it's much easier to introduce new features.

The new .NET Compiler Platform - Roslyn

Since the beginning of time, we've considered compilers as black boxes, a mere piece of the toolchain that transforms code into something that can be executed (and hopefully, that works as expected). This way of picturing a compiler was fine in the past, but it's no longer suitable for modern days. If you're a .NET developer, you probably know of Visual Studio features like Go to Definition, Smart Rename, and so on. These are all powerful refactoring and code analysis tools that help us on a daily basis to improve the quality of our code. As these tools get smarter, they need access to more and more of the deep code knowledge that only compilers possess.

Thanks to Roslyn (the codename for the new .NET Compiler Platform), we can leverage in an "As-a-Service" fashion a set of APIs that are able to communicate directly with the compiler, allowing tools and end users to share in the wealth of information compilers have about our code. The transition to compilers as platforms dramatically lowers the barrier to entry for creating code-focused tools and applications.

Roslyn's API layers

Roslyn consists of two main layers of APIs: the Compiler APIs and Workspaces APIs. The compiler layer contains the object models that correspond with information exposed at each phase of the compiler pipeline, both syntactic and semantic. The compiler layer also contains an immutable snapshot of a single invocation of a compiler, including assembly references, compiler options, and source code files. The Workspaces layer contains the Workspace API, which is the starting point for doing code analysis and refactoring over entire solutions. This layer has no dependencies on Visual Studio components. In fact, even Visual Studio Code, the free and X-Platform Visual Studio-like IDE, uses Roslyn to provide a rich development experience to C# developers.

Note: If you want to learn more about Visual Studio Code, you can visit the official website. You can also download the free e-book about Visual Studio Code from Syncfusion's Succinctly series.

.NET Native

.NET Native is a pre-compilation technology for building modern apps in Visual Studio 2015. The .NET Native toolchain will compile your managed IL binaries into native binaries. Every managed (C# or VB) Universal Windows app will utilize this new technology.

For users of your apps, .NET Native offers these advantages:

- Fast execution times
- Consistently speedy startup times
- Low deployment and update costs
- Optimized app memory usage

.NET Native is able to bring the performance benefits of C++ to managed code developers because it uses the same or similar tools as C++ under the hood.

What is .NET Core?

.NET Core 1.0 is a new runtime, modular and enriched with a subset of the API part of the .NET Framework. We have a feature-complete product on Windows, while on the other platforms (Linux and OSX), there are still features under development. .NET Core 1.0 can be divided into two major sections: one called CoreFX, which consists of a small set of libraries, and one called CoreCLR, and a small and optimized runtime.

NET Core is one of Microsoft's projects under the stewardship of the .NET Foundation, meaning that it's open source, and we can all contribute to it and follow its progress. Later in the book we'll dive into the requirements that we have to meet if we want to contribute to the project.

In pursuit of modularity, Microsoft chose to distribute the CoreCLR runtime and the CoreFX libraries on NuGet, factoring them as individual NuGet packages. The packages are named after their namespace in order to facilitate their discovery during search.

One of the key benefits of .NET Core is its portability. You can package and deploy the CoreCLR with your application, eliminating your application's dependency on an installed version of .NET. You can host multiple applications side-by-side using different versions of the CoreCLR, and upgrade them individually, rather than being forced to upgrade all of them simultaneously. CoreFX has been built as a componentized set of libraries, each requiring the minimum set of library dependencies. This approach enables minimal distributions of CoreFX libraries (just the ones you need) within an application, alongside CoreCLR.

Note: .NET Core is not intended to be smaller than the .NET Framework, but thanks to this modular-focused model, it allows applications to depend only on libraries they need, hence allowing for a smaller memory footprint.

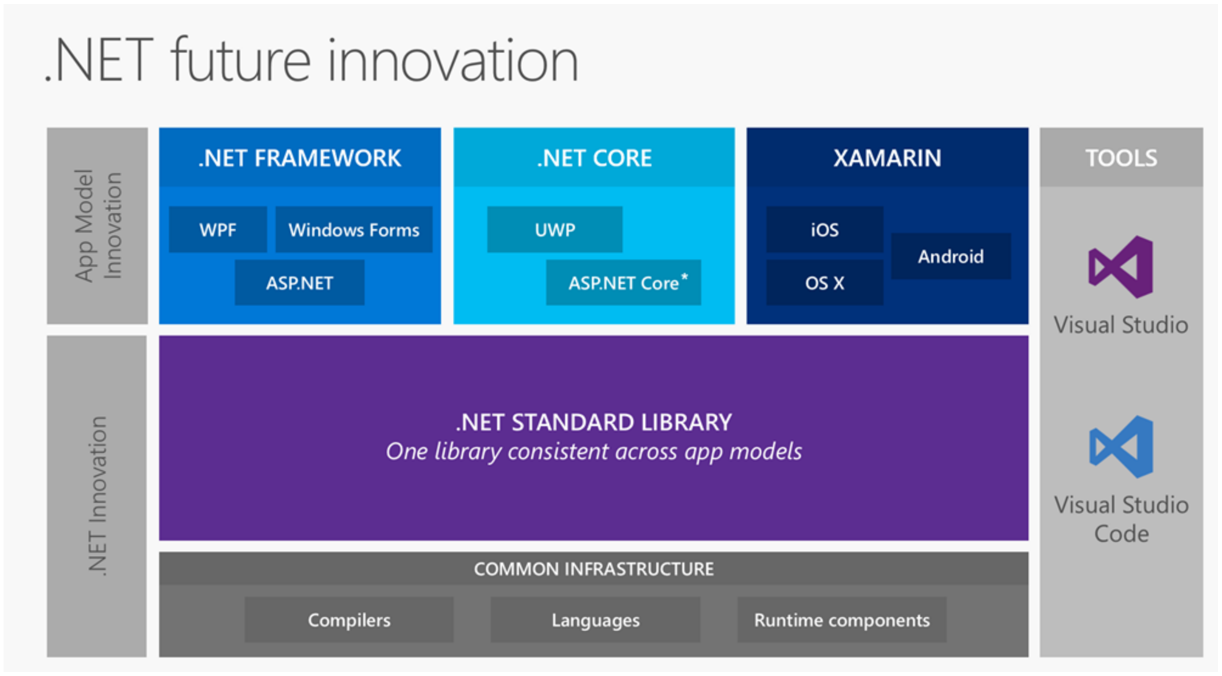


Figure 2: A diagram that summarizes .NET Core

Figure 2 expands the one we saw before by adding a column that lists the tools we as developers have at our disposal to consume the different frameworks. In regards to .NET Core, we can see that besides ASP.NET Core, it includes another workload, the Universal Windows Platform. Previously I mentioned the .NET Standard Library, but I've basically skipped any sort of clarification of what it actually is. It's an important topic that's very hard to summarize and simplify. If you'd like to read more about it, head over to the official documentation on GitHub.

What about the full .NET Framework?

If you've read this far, and you're a bit paranoid like me, you have certainly wondered, "Where will the full .NET Framework end up after all this *NuGet-Cross-Modular extravaganza*?" Fear not—the.NET Framework is still the platform of choice for building rich desktop applications, and .NET Core doesn't change that.

However, now that Visual Studio 2015 is out, .NET Core will version faster than the full framework, meaning that sometimes some features will only be available on .NET Core-based platforms. The full framework will still be updated constantly, bringing in, when possible, the innovative concepts and features of .NET Core.

Of course, the team's goal would be to minimize API and behavioral differences between the two, but also not to break compatibility with existing .NET Framework applications. There are also investments that are exclusively being made for the .NET Framework, such as the work the team announced in the WPF Roadmap.

The role of Mono

For those who don’t know Mono, it’s essentially an open source re-implementation of the .NET Framework. As such, it shares the richness of the APIs with the .NET Framework, but it also shares some of its problems, specifically around the implementation factoring. Another way to look at it: The .NET Framework has essentially two forks. One fork is provided by Microsoft and is Windows-only. The other fork is Mono, which you can use on Linux and Mac.

Mono won’t be discussed in detail during the course of this e-book, but whenever possible, I’ll try to point you to some useful resource that might help you learn more about it.

Chapter 1 The OSS Strategy behind .NET Code

As I said in the previous chapter, .NET Core is open source. At the time of the announcement (October 31, 2016), this was a huge day in the history of Microsoft, and .NET in particular. The .NET team decided to open-source .NET Core mainly in order to leverage an enormous ecosystem of developers, and to lay the foundations for a cross-platform .NET.

In terms of laying the foundations for being cross platform, the goal was to unify the divided code base of Mono and the Windows implementation of the .NET Framework. This improves the developer experience by making a single cross-platform stack, allowing anyone to contribute. In terms of leveraging a broader audience of developers, it should be seen as the willingness of Microsoft to go where the developers are, and not to pretend to be followed blindly by new developers without any concrete reason.

One of the core tools, or platforms if you prefer, of these strategies has been GitHub, because it seems the majority of the .NET community is on GitHub.

Just like any other .NET project, the .NET team accepts contributions in what are open issues, “up-for-grabs” or simply new features that you, as a day-by-day user, may think are useful. Of course, PR are not automatically accepted; instead, they are reviewed and judged based on the following two criteria:

- **Roadmap**. All projects focus their energy on certain areas. In order to keep the focus and the momentum, it’s important that most of the work is aligned with the product’s roadmap.
- **Quality**. External folks have to meet the same quality bar that Microsoft employees have to meet. This includes having the right design, architecture, sufficient test coverage, and following the coding style.

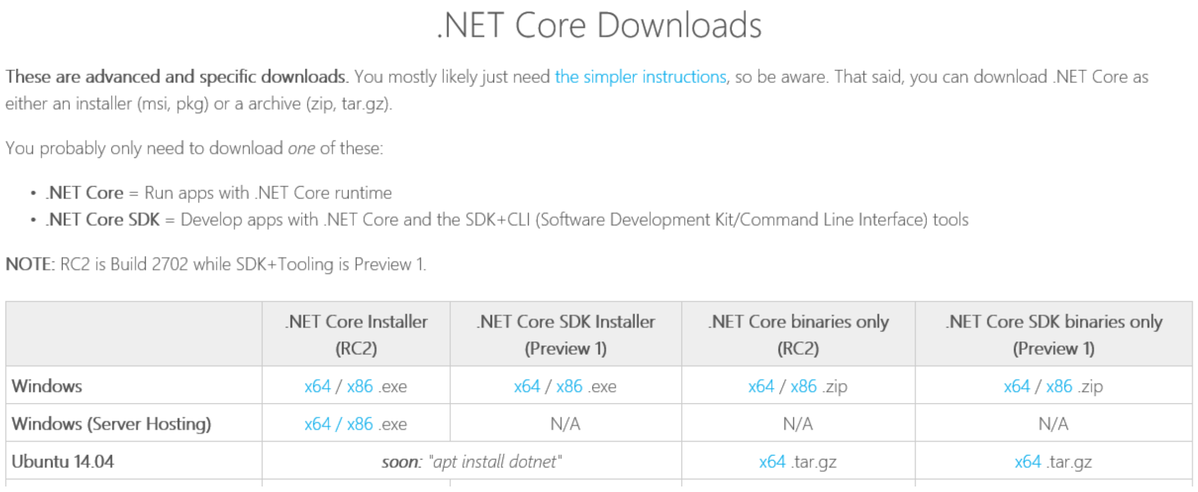
The .NET Foundation

The .NET Core project is under the stewardship of the .NET Foundation. The .NET Foundation is an independent organization that fosters open development and collaboration across the Microsoft .NET development platform. The goal of the .NET Foundation is to promote openness and community participation, as well as encouraging innovation. You can read more here about .NET Core and its role inside the .NET Foundation.

Chapter 2 Playing with .NET Core

Installing .NET Core on Windows

Before we start to play with .NET Core, we must install it. The easiest way to do it is to visit https://www.microsoft.com/net/core and download the official MSI installer.



.NET Core Downloads

These are advanced and specific downloads. You mostly likely just need the simpler instructions, so be aware. That said, you can download .NET Core as either an installer (msi, pkg) or a archive (zip, tar.gz).

You probably only need to download *one* of these:

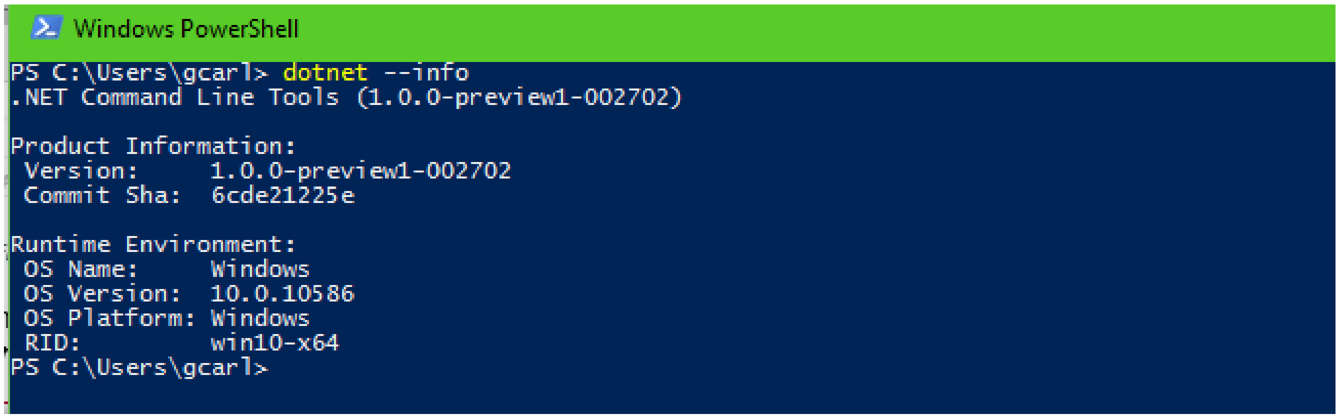
- .NET Core = Run apps with .NET Core runtime
- .NET Core SDK = Develop apps with .NET Core and the SDK+CLI (Software Development Kit/Command Line Interface) tools

NOTE: RC2 is Build 2702 while SDK+Tooling is Preview 1.

	.NET Core Installer (RC2)	.NET Core SDK Installer (Preview 1)	.NET Core binaries only (RC2)	.NET Core SDK binaries only (Preview 1)
Windows	x64 / x86 .exe	x64 / x86 .exe	x64 / x86 .zip	x64 / x86 .zip
Windows (Server Hosting)	x64 / x86 .exe	N/A	N/A	N/A
Ubuntu 14.04	*soon: "apt install dotnet"*		x64 .tar.gz	x64 .tar.gz

Figure 3: Installing .NET Core - Step 1

The installation process is straightforward, and once finished, we should be able to fire up PowerShell, type **`dotnet --info`**, and get the following output:



```
PS C:\Users\gcarl> dotnet --info
.NET Command Line Tools (1.0.0-preview1-002702)

Product Information:
 Version:     1.0.0-preview1-002702
 Commit Sha:  6cde21225e

Runtime Environment:
 OS Name:     Windows
 OS Version:  10.0.10586
 OS Platform: Windows
 RID:         win10-x64
PS C:\Users\gcarl>
```

Figure 4: Successful .NET Core installation

The .NET Core CLI (Command line interface)

The CLI comes with a series of common commands:

Common options (passed before the command)

Table 1: CLI's common options

Command Syntax	Description
`-v | --verbose`	Enable verbose output
`--version`	Display .NET CLI Version Info

Common commands

Table 2: CLI's common commands

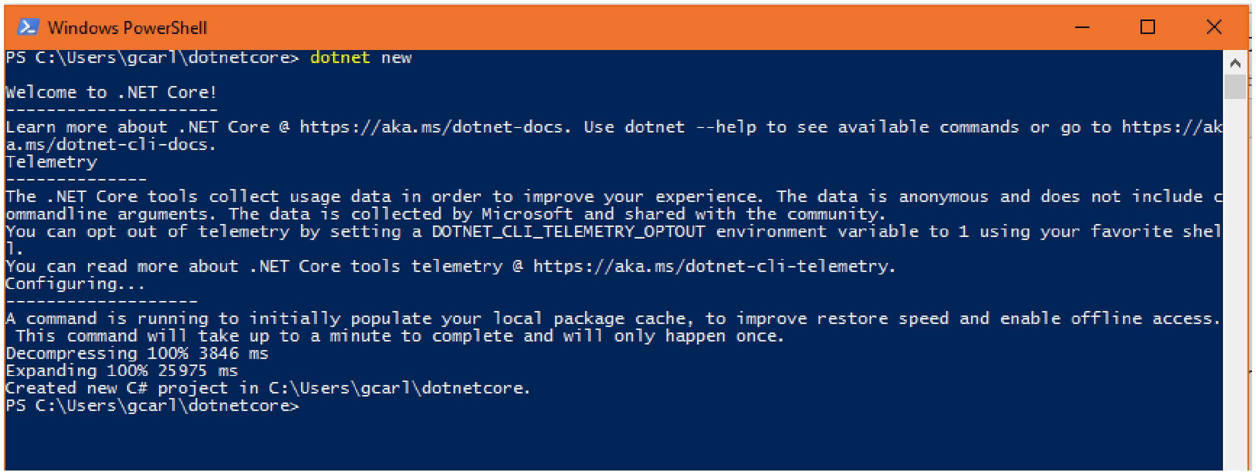
Command Syntax	Description
`new`	Initialize a basic .NET project
`restore`	Restore dependencies specified in the .NET project
`build`	Build a .NET project
`publish`	Publish a .NET project for deployment (including the runtime)
`run`	Compile and immediately execute a .NET project
`test`	Run unit tests using the test runner specified in the project
`pack`	Create a NuGet package

Tip: This website automatically detects the version of your OS and selects the right setup option for you.

Creating a project with the .NET Core CLI

As we saw in the previous paragraph, thanks to the CLI, creating a new .NET Core application seems quite easy. In fact, it is easy. In order to try this out and start playing a bit with .NET Core, let's open up PowerShell or the command prompt and navigate to a folder of your choice. In my case, I simply created a folder named **dotnetcore** in my User folder.

Once you feel like you are ready to go, type the following command: **dotnet new**. You should receive an output similar to the following.

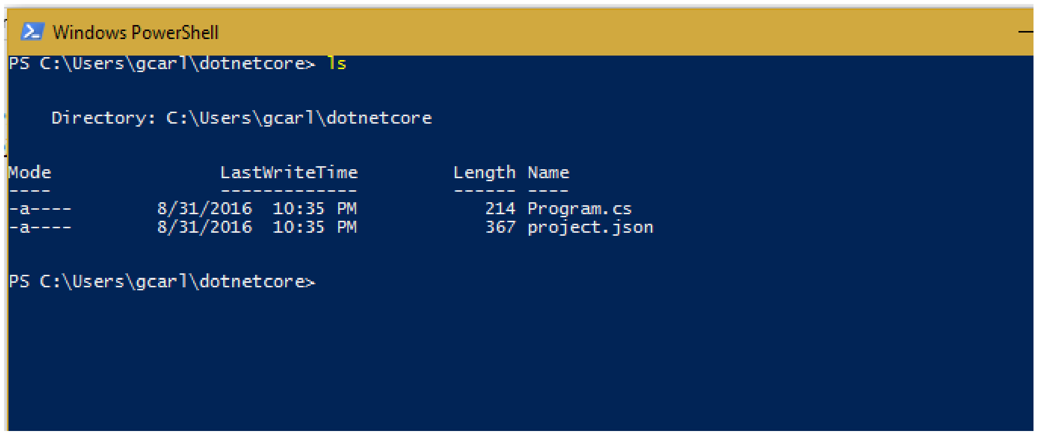


```
PS C:\Users\gcarl\dotnetcore> dotnet new

Welcome to .NET Core!
---------------------
Learn more about .NET Core @ https://aka.ms/dotnet-docs. Use dotnet --help to see available commands or go to https://ak
a.ms/dotnet-cli-docs.
Telemetry
--------------
The .NET Core tools collect usage data in order to improve your experience. The data is anonymous and does not include c
ommandline arguments. The data is collected by Microsoft and shared with the community.
You can opt out of telemetry by setting a DOTNET_CLI_TELEMETRY_OPTOUT environment variable to 1 using your favorite shel
l.
You can read more about .NET Core tools telemetry @ https://aka.ms/dotnet-cli-telemetry.
Configuring...
-------------------
A command is running to initially populate your local package cache, to improve restore speed and enable offline access.
 This command will take up to a minute to complete and will only happen once.
Decompressing 100% 3846 ms
Expanding 100% 25975 ms
Created new C# project in C:\Users\gcarl\dotnetcore.
PS C:\Users\gcarl\dotnetcore>
```

Figure 5: Successful project creation through the CLI

If this is the first time you run the **dotnet new** command, the CLI will populate your local package folder so that the operation of restoring NuGet packages is faster. The CLI also warns you that since the tooling is still in preview, a lot of telemetry is collected. If you're not comfortable with that, you can turn telemetry off. You can find more information about telemetry at https://aka.ms/dotnet-cli-telemetry.

This simple command created three files inside the current directory. If you happen to be playing with ASP.NET Core 1.0, you might be familiar with the one named **project.json**. If not, don't worry—we will now explain the purpose of each one.


Windows PowerShell

```
PS C:\Users\gcarl\dotnetcore> ls


    Directory: C:\Users\gcarl\dotnetcore


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        8/31/2016  10:35 PM            214 Program.cs
-a----        8/31/2016  10:35 PM            367 project.json


PS C:\Users\gcarl\dotnetcore>
```

Figure 6: The list of files created by the new command

Project.json

The Project.json file is used in every project that uses NuGet 3.0, and it serves as a configuration file for both NuGet and Visual Studio. Inside it, we can specify our dependencies, reference frameworks, custom commands, and compile options. Starting from Visual Studio 2015, several project types are utilizing this technology, such as:

- Universal Windows Platform managed apps (UWP)
- Portable class libraries (PCL)
- ASP.NET Core applications

Further information about this file can be found in the official NuGet documentation or in the ASP.NET Core 1.0 documentation (available on GitHub). The following is the content of the Project.json file generated by the .NET CLI:

```
{
  "version": "1.0.0-*",
  "buildOptions": {
    "debugType": "portable",
    "emitEntryPoint": true
  },
  "dependencies": {},
  "frameworks": {
    "netcoreapp1.0": {
      "dependencies": {
        "Microsoft.NETCore.App": {
          "type": "platform",
          "version": "1.0.1"
        }
      },
      "imports": "dnxcore50"
    }
  }
}
```

As you can see, we are building our console application against the .NET Core Framework (identified by the TFM **netcoreapp1.0**), and we only have one dependency, **Microsoft.NETCore.App**. The full schema of the Project.json file can be found here.

Note: At the time of writing, the .NET team made an announcement about some changes to he project.json concepts. You can read more about it here.

Program.cs

This is easiest file to understand; it's a basic C# file with a "**Hello World**" code snippet in it.

```
using System;

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

Transitioning from DNVM to the .NET CLI

Before starting to play with the CLI and compiling and running our .NET Core console application, we must first ensure that our environment is configured correctly. In order to do that, we must check to see if the **DNVM** (acronym of .NET Version Manager) is installed, and that we have the latest (in our case, the unstable) version of the runtime installed and set as active.

As its name implies, DNVM provides the functionality needed to configure your .NET runtime. We can use DNVM to specify which version of the DNX (.NET Execution Environment) to use at the process, user, or machine level.

The .NET Execution Environment (DNX) is a software development kit (SDK) and runtime environment that has everything you need to build and run .NET applications for Windows, Mac, and Linux. It provides a host processes, CLR hosting logic, and managed entry-point discovery. DNX was built for running cross-platform ASP.NET Web applications, but it can run other types of .NET applications, too, such as cross-platform console apps.

Checking where the DNVM utility is installed

When the .NET Team released the RC1 version of .NET Core and ASP.NET Core, they also announced DNX tooling. However, with RC2 they transitioned to the newly introduced .NET Core CLI. They had their reasons, one being that the set of features of this toolset was made redundant by the CLI.

Scott Hanselman wrote an amazing blog post when the transition was happening. Reading it may help you understand this complex decision better.

For those who are completely new the existence of this set of tools (DNX, DNVM, and DNU), perhaps only because you've just started working with .NET Core, all you need to know is that DNX was a runtime and a toolset used to build .NET Core and ASP.NET Core applications. It was divided into three main pieces:

- DNVM (Dotnet Version Manager): Used to download and manage the runtimes in your machine
- DNX (Dotnet Execution Runtime): The runtime that executes your code
- DNU (Dotnet Developer Utility): Used to package and publish your app and managing dependencies

Now that .NET Core is an RTM version, the tools, which are still in preview, that we have available are the CLI interface and Visual Studio or Visual Studio Code. The .NET CLI comes in two flavors: It can be installer-specific for the development platform you are using, or it can be an install script, useful for scenarios such as a continuous integration server.

One of the differences between the two toolsets is that now, in order for you to target a specific version of the framework, you need to add a package of a certain version, whereas with the DNX toolset, that came free with the runtime selection features of DNVM.

Commands not available in the CLI that were available in DNX

With the change in the toolset, some commands were remapped, and some were removed. Basically, all the commands now start with **dotnet** instead of **dnx** or **dnu.** If you already learned where a specific command was, you now have to simply replace that with **dotnet.** Table 3 contains a full list of the commands that were removed in the CLI.

Table 3: Commands not implemented in the CLI

DNX Command	CLI Command	Description
Dnx [cmd-name]	N/A	In DNX, run a command as defined in project.json.
Dnu install	N/A	In DNX, install a package as a dependency.
Dnu wrap	N/A	In DNX, wrap project.json in csproj.
Dnu commands	N/A	In DNX, manage the globally installed commands.

More about the transition

The .NET team had a valid reason for deciding not to support those commands in the CLI. However, since this transition is not always painless, there are some workarounds in case you still need to leverage those mechanisms in the latest version of the tooling.

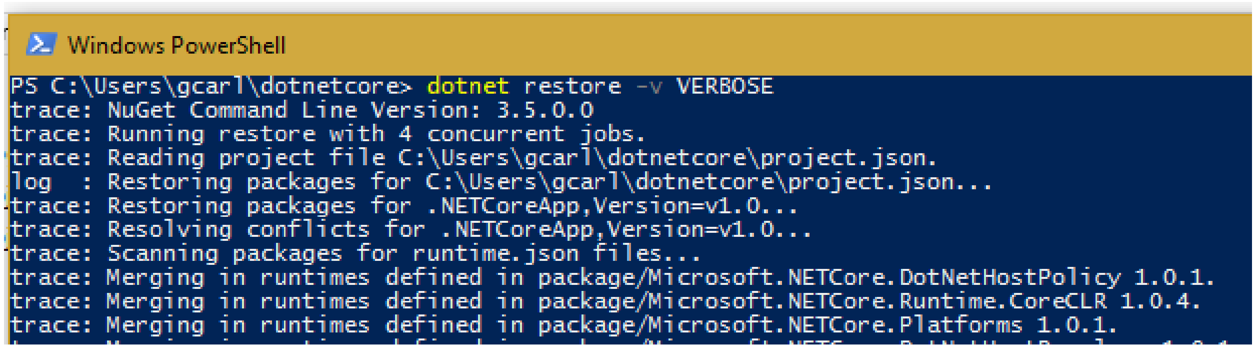
The **Global Commands** are no longer supported because they were essentially a console application packaged as a NuGet package that was then invoked by DNX. The CLI is unfamiliar with this concept. However, you can still use the **dotnet <command>** syntax to achieve the same result.

With the CLI, the **install** command isn't needed. In order to install a dependency, you need to edit the project.json file and run **dotnet restore**. Finally, in order for you to run your code, you can use the **dotnet run** command from the same directory where you previously did **dotnet new** or you can specify the path to your compiled assembly to the **dotnet** command, such as **dotnet assembly/debug/compiled.dll**

The transition from the DNX toolset to the CLI also had an impact on the way a project is structured. For this reason, the team set up a nice how-to that demonstrates migrating a DNX project to a CLI project. You can read more about it at https://docs.microsoft.com/en-us/dotnet/articles/core/migrating-from-dnx#migrating-your-dnx-project-to-net-core-cli.

Compile and run a project with the .NET CLI interface

In order to compile and then run our basic console application, we must first restore the dependencies that we have previously described. In order to do that, we simply need to type in the command `dotnet restore`, and the .NET CLI should start restoring our packages. I suggest you to use the `-v` flag enable Verbose logging.

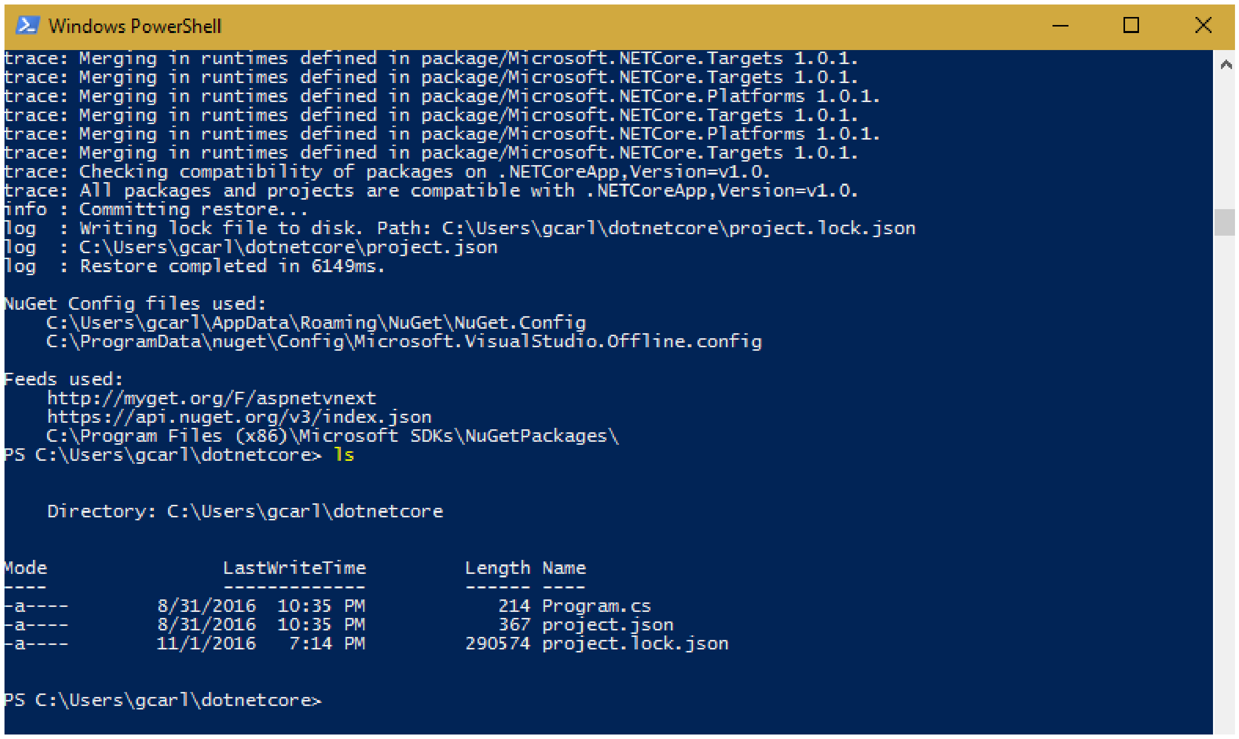


Windows PowerShell

```
PS C:\Users\gcarl\dotnetcore> dotnet restore -v VERBOSE
trace: NuGet Command Line Version: 3.5.0.0
trace: Running restore with 4 concurrent jobs.
trace: Reading project file C:\Users\gcarl\dotnetcore\project.json.
log  : Restoring packages for C:\Users\gcarl\dotnetcore\project.json...
trace: Restoring packages for .NETCoreApp,Version=v1.0...
trace: Resolving conflicts for .NETCoreApp,Version=v1.0...
trace: Scanning packages for runtime.json files...
trace: Merging in runtimes defined in package/Microsoft.NETCore.DotNetHostPolicy 1.0.1.
trace: Merging in runtimes defined in package/Microsoft.NETCore.Runtime.CoreCLR 1.0.4.
trace: Merging in runtimes defined in package/Microsoft.NETCore.Platforms 1.0.1.
```

Figure 7: The output of the dotnet restore command

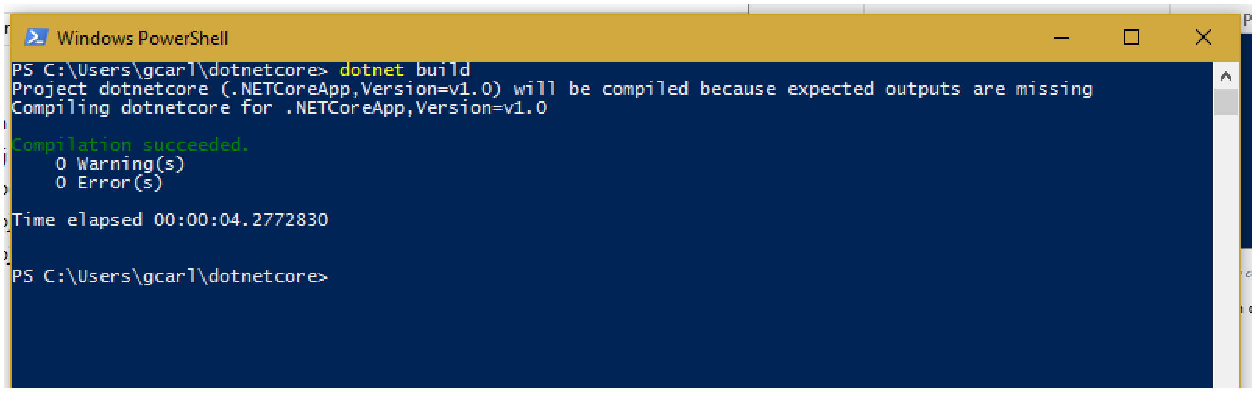
Once the restore process is complete, you should have a new file named **project.lock.json** in the root of your project. You don't need to know all the internals of this file; however, we'll talk more about this file shortly.



```
trace: Merging in runtimes defined in package/Microsoft.NETCore.Targets 1.0.1.
trace: Merging in runtimes defined in package/Microsoft.NETCore.Targets 1.0.1.
trace: Merging in runtimes defined in package/Microsoft.NETCore.Platforms 1.0.1.
trace: Merging in runtimes defined in package/Microsoft.NETCore.Targets 1.0.1.
trace: Merging in runtimes defined in package/Microsoft.NETCore.Platforms 1.0.1.
trace: Merging in runtimes defined in package/Microsoft.NETCore.Targets 1.0.1.
trace: Checking compatibility of packages on .NETCoreApp,Version=v1.0.
trace: All packages and projects are compatible with .NETCoreApp,Version=v1.0.
info : Committing restore...
log  : Writing lock file to disk. Path: C:\Users\gcarl\dotnetcore\project.lock.json
log  : C:\Users\gcarl\dotnetcore\project.json
log  : Restore completed in 6149ms.

NuGet Config files used:
    C:\Users\gcarl\AppData\Roaming\NuGet\NuGet.Config
    C:\ProgramData\nuget\Config\Microsoft.VisualStudio.Offline.config

Feeds used:
    http://myget.org/F/aspnetvnext
    https://api.nuget.org/v3/index.json
    C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\
PS C:\Users\gcarl\dotnetcore> ls


    Directory: C:\Users\gcarl\dotnetcore


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        8/31/2016  10:35 PM            214 Program.cs
-a----        8/31/2016  10:35 PM            367 project.json
-a----        11/1/2016   7:14 PM         290574 project.lock.json


PS C:\Users\gcarl\dotnetcore>
```

Figure 8: The folder structure after the dnu restore command

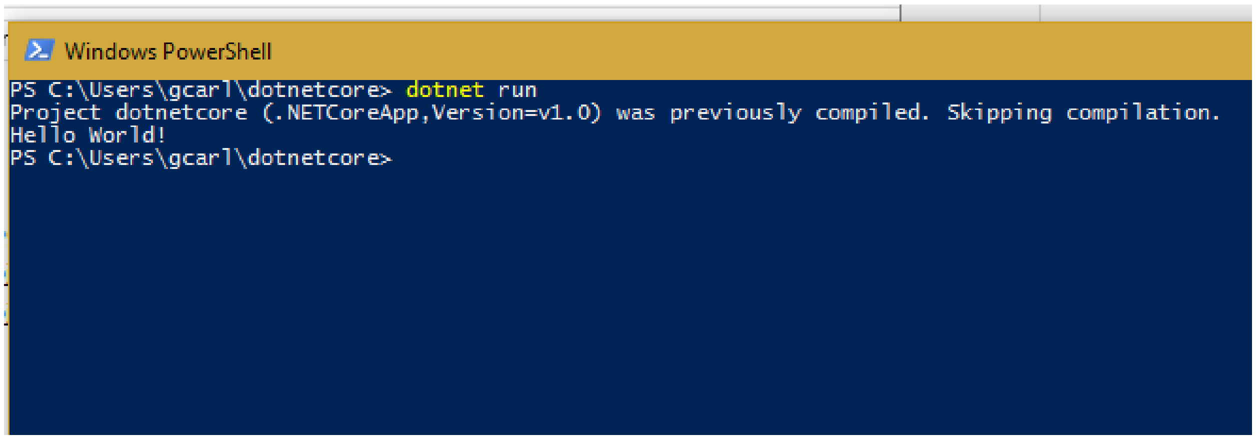
Now we are ready to compile our program using the **`build`** option of the CLI.



```
PS C:\Users\gcarl\dotnetcore> dotnet build
Project dotnetcore (.NETCoreApp,Version=v1.0) will be compiled because expected outputs are missing
Compiling dotnetcore for .NETCoreApp,Version=v1.0

Compilation succeeded.
    0 Warning(s)
    0 Error(s)

Time elapsed 00:00:04.2772830


PS C:\Users\gcarl\dotnetcore>
```

Figure 9: Output of a successful compilation

If we want to run our application, we simply have to type **`dotnet run`**, and if we did everything correctly, the output you get should be like the following:



```
PS C:\Users\gcarl\dotnetcore> dotnet run
Project dotnetcore (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
Hello World!
PS C:\Users\gcarl\dotnetcore>
```

Figure 10: A Hello World message from .NET Core

Of course, in a realistic example we would have edited the **Program.cs** file with some more advanced code (maybe an HTTP request), but this is a good starting point. In the course of the book, we will build a more advanced console application that will leverage all the further knowledge we will gain about the .NET CLI interface.

The project.lock.json file, AKA the lock file

The project.lock.json file is what now can be considered a solution-level packages folder. It basically tells us what set of packages should be used at compile time by our application. It's a way to scope the list of packages in **%userprofile%.dnx\packages** so that projects run considering only the packages relevant to the project. The second purpose of the lock file is to store the list of files and relevant content for compilation and runtime so that the runtime only has to read a single file instead of many .nuspec files. In an ASP.NET Core workload on Azure, this cuts the startup time in half. The following code is a portion of our application lock file.

```
{
  "locked": false,
  "version": 2,
  "targets": {
    ".NETCoreApp,Version=v1.0": { /* many packages */}
  },
  "libraries": { /* many libraries */ },
  "projectFileDependencyGroups": {
    "": [],
    ".NETCoreApp,Version=v1.0": [
      "Microsoft.NETCore.App >= 1.0.1"
    ]
  },
  "tools": {},
  "projectFileToolGroups": {}
}
```

As always, the structure is defined using a JSON schema. Let's analyze each node. At the top of the file is a "locked" property. The "locked" property is a Boolean, and therefore supporting two schools of thought. However, we need to understand the "target" node before we can see the effect the locked property has.

The **target** node holds the list of all the packages that the app is using. One thing to notice is how it differs from the list inside the "unlocked" project.json file. Under each package node we have a list of all its dependencies and the path to the reference assembly and the runtime assembly.

Another difference from project.json is that here we can't use wildcards and be vague about versioning. This small but important distinction is made to support two different schools of thoughts about checking in dependencies into source control. The first is to lock the dependencies, setting the **`locked`** property to **`true`** to avoid any broken compilation due to API changes. The second school of thought is to leave the lock file unlocked, with the **`locked`** property set to **`false`**, so that the restore process can pick the latest version of the dependency (having as an upper limit the ones defined in the project.json file). If you find yourself in the position of deciding between them, just know that if you use the default Visual Studio .gitignore file on GitHub today, project.lock.json is specified to not be committed to source control.

Side note on reference assemblies

Quoting Jared Parsons, Developer on the C# Compiler: "A reference assembly is a slimmed down version of an implementation assembly that contains the API surface but no real code. A program can reference these assemblies at compile time, but cannot run against them. Instead, at deploy time, programs are paired with the original implementation assembly. Breaking up assemblies into reference and implementation pairs is a useful tool for creating targeted API surfaces." You can read more about Jared's adventure in his blog.

Advanced settings for the .NET Core CLI commands

You might have noticed that in the previous paragraph, during the package restore step, we specified a **`-v`** parameter to enable more verbose logging. This should be enough to raise a series of more than justified questions: Are there any more options like that? And if there are, what commands are available? What do they do? How do I use them?

Luckily for you, discovering these options is quite easy; you just need to append the **`-h`** flag after the option name. Here's an example:

```
PS F:\dotnetcore> dotnet <command> -h
```

The following tables show all the available flags for each command. When a command is omitted, it simply means that it does not have any flags available.

Command options for: dotnet restore

Option	Description
`--force-english-output`	Forces the application to run using an invariant, English-based culture
`-s <source>`	Specifies a NuGet package source to use during the restore
`--packages <pkgDir>`	Directory to install packages in
`--disable-parallel`	Disables restoring multiple projects in parallel
`-f <feed>`	A list of package sources to use as a fallback
`--configfile <file>`	The NuGet configuration file to use
`--no-cache`	Do not cache packages and HTTP requests
`--infer-runtimes`	Temporary option to allow NuGet to infer RIDs for legacy repositories
`-v <verbosity>`	The verbosity of logging to use. Allowed values: Debug, Verbose, Information, Warning, Error
`--ignore-failed-sources`	Only warning of failed sources if there are packages meeting version requirements

Command options for: dotnet build

Option	Description
`-o <OUTPUT_DIR>`	Directory in which to place outputs
`-b <OUTPUT_DIR>`	Directory in which to place temporary outputs
`-f <FRAMEWORK>`	Compile a specific framework
`-r <RUNTIME_IDENTIFICATION>`	Target runtime to publish for
`-c <CONFIGURATION>`	Configuration under which to build
`--version-suffix`	Defines what * should be replaced with in version field in project.json
`--build-profile`	Set this flag to print the incremental safety checks that prevent incremental compilation
`--no-incremental`	Set this flag to turn off incremental build
`--no-dependencies`	Set this flag to ignore project-to-project references and only build the root project

Command options for: dotnet publish

Option	Description
`-f <FRAMEWORK>`	Target framework to compile for
`-r <RUNTIME_IDENTIFIER>`	Target runtime to publish for
`-b <OUTPUT_DIR>`	Directory in which to place temporary outputs

Option	Description
`-o <OUTPUT_PATH>`	Path in which to publish the app
`-c <CONFIGURATION>`	Configuration under which to build
`--native-subdirectory`	Temporary mechanism to include subdirectories from native assets of dependency packages in output
`--version-suffix <VERSION_SUFFIX>`	Defines what * should be replaced with in version field in project.json
`--no-build`	Do not build projects before publishing

Command options for: dotnet run

Option	Description
`-f <arg>`	Compile a specific framework
`-c <arg>`	Configuration under which to build
`-p <arg>`	The path to the project to run (defaults to the current directory). Can be a path to a project.json or a project directory
`<args>`	Arguments to pass to the executable or script

Command options for: dotnet pack

Option	Description
`-b <OUTPUT_DIR>`	Directory in which to place temporary build outputs
`-o <OUTPUT_DIR>`	Directory in which to place outputs
`-c <CONFIGURATION>`	Configuration under which to build
`--version-suffix <VERSION_SUFFIX>`	Defines what * should be replaced with in “version” field in project.json
`--no-build`	Do not build project before packing
-s | --serviceable	Set the serviceable flag in the package

Command options for: dotnet test

Option	Description
`--parentProcessId`	Used by IDEs to specify their process ID. Test will exit if the parent process does.
`--port`	Used by IDEs to specify a port number to listen for a connection
`-c  <CONFIGURATION>`	Configuration under which to build
`-o <OUTPUT_DIR>`	Directory in which to find the binaries to be run
`-b <OUTPUT_DIR>`	Directory in which to find temporary outputs
`-f  <FRAMEWORK>`	Look for test binaries for a specific framework

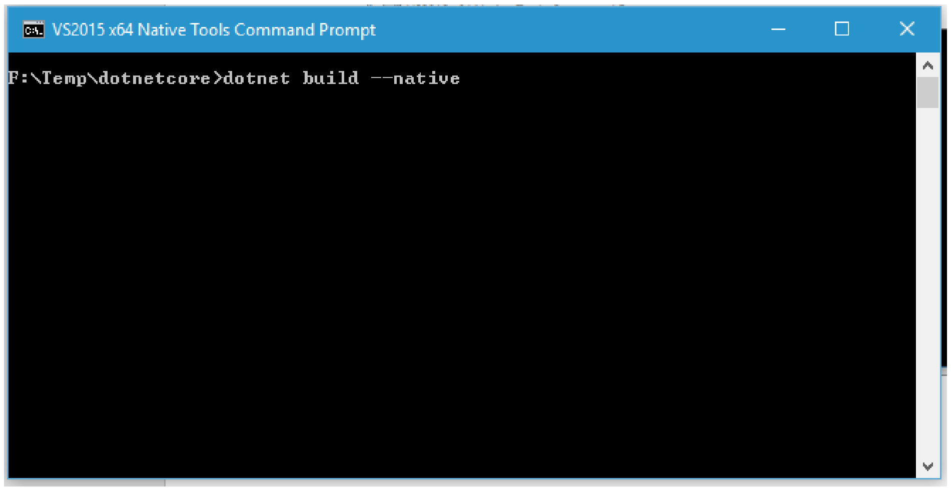
Option	Description
`-r <RUNTIME_IDENTIFIER>`	Look for test binaries for a for the specified runtime
`--no-build`	Do not build project before testing

Compile a native version of our console application

Now that we have seen the wealth of options we have in our commands, let's try one out. We will now build our console application, specifying the **`-n`** flag on the **`compile`** command. This will prompt the .NET CLI to run the compilation through the .NET Native toolchain. The .exe that pops out is bigger, but it is only a single file.

Tip: This option is no longer available in RC2. However, if you download any version of .NET Core prior to RC2, you should be able to follow along. You can find more information about this on GitHub.

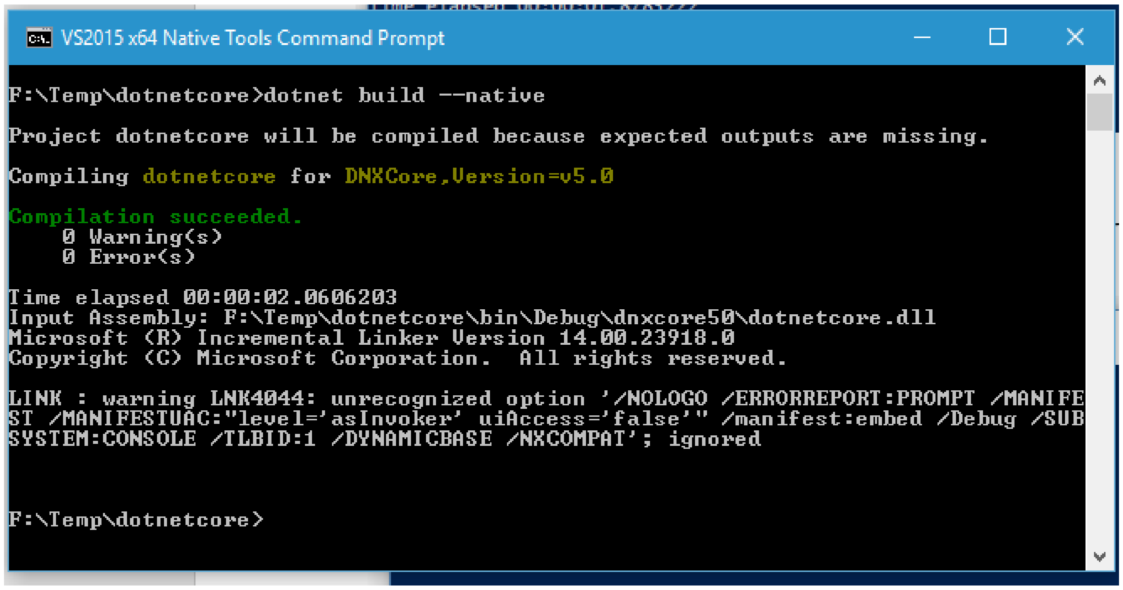
In order to compile successfully with the **`--native`** flag, we must run the compilation from the **VS2015 x64 Native Command Tools Command Prompt.**



```
F:\Temp\dotnetcore>dotnet build --native
```

Figure 11: The VS2015 x64 Native Command Line

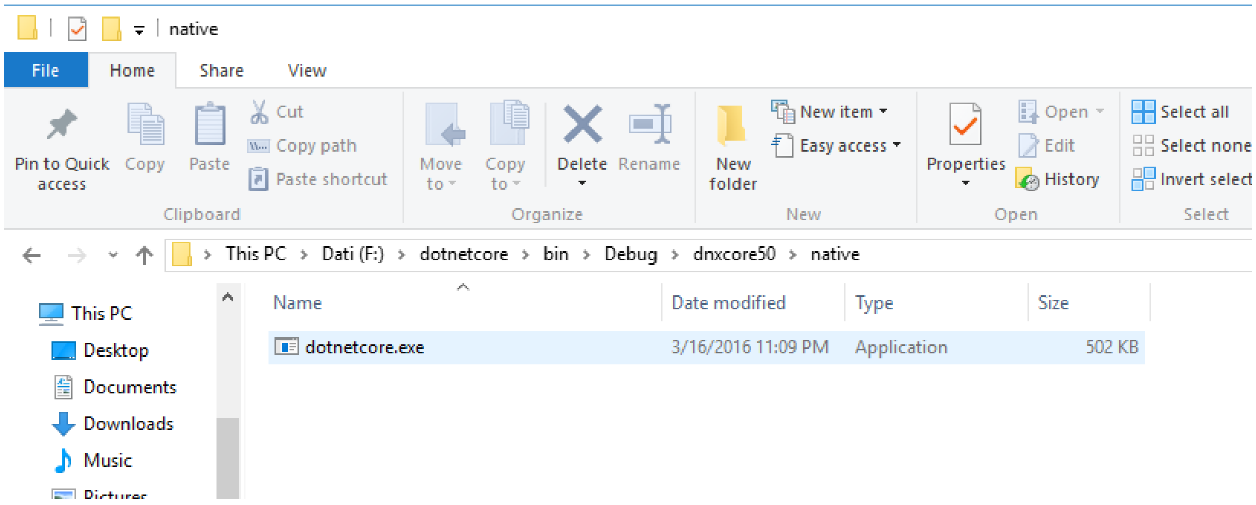
If you hit **Enter**, you should get this output (ignore the warning):



```
F:\Temp\dotnetcore>dotnet build --native

Project dotnetcore will be compiled because expected outputs are missing.

Compiling dotnetcore for DNXCore,Version=v5.0

Compilation succeeded.
    0 Warning(s)
    0 Error(s)

Time elapsed 00:00:02.0606203
Input Assembly: F:\Temp\dotnetcore\bin\Debug\dnxcore50\dotnetcore.dll
Microsoft (R) Incremental Linker Version 14.00.23918.0
Copyright (C) Microsoft Corporation.  All rights reserved.

LINK : warning LNK4044: unrecognized option '/NOLOGO /ERRORREPORT:PROMPT /MANIFE
ST /MANIFESTUAC:"level='asInvoker' uiAccess='false'" /manifest:embed /Debug /SUB
SYSTEM:CONSOLE /TLBID:1 /DYNAMICBASE /NXCOMPAT'; ignored


F:\Temp\dotnetcore>
```

Figure 12: The console output of the native compilation

And of course, if we look into the output folder **F:\dotnetcore\bin\Debug\dnxcore50\native**, we should have only one file, our executable (notice how the size is slightly higher because of the native compilation).



Name	Date modified	Type	Size
dotnetcore.exe	3/16/2016 11:09 PM	Application	502 KB

Figure 13: Our native executable

Chapter 3 Contributing to the .NET Core repository

The .NET Core project has been developed and is now being maintained in an open source fashion on GitHub. However, just like every project, there are some coding rules and a code of conduct that you must follow in order to be a one of the “good citizens.” In this chapter, we will cover the coding guidelines that you must follow if you want to contribute to the project.

PR guidelines

The team, at some point in time, will re-style the complete codebase of the project according to the coding guidelines (which we’ll we talk about later in this chapter). In the meantime, you should:

- **NOT** send PRs for style changes. For example, do not send PRs renaming all the occurrences of Int32 to int.
- **NOT** send PRs for upgrading code to use newer language features. Those features can be used in new code, but for the existing code, it is a whole new thing.
- **GIVE** priority to the current style of the project or file you're changing even if it diverges from the general guidelines.
- **NOT** submit to the *master* branch API additions to any type that has shipped in the full .NET framework. And in addition, do not submit PR to the APIs that have not yet been approved.
- **INCLUDE** tests when adding new features. When fixing bugs, start with adding a test that highlights how the current behavior is broken.
- **KEEP** the discussions focused.
- **NOT** surprise the team with big pull requests. Instead, file an issue and start a discussion so that the team can help you with your developments.
- **NOT** commit code that you didn't write. If you find code that you think is a good fit to add to .NET Core, file an issue and start a discussion before proceeding.
- **NOT** submit PRs that alter licensing-related files or headers.

Coding guidelines: C#, C++, and general code files

The general rule is to follow Visual Studio defaults. However, there are some more detailed cases. For C++ files (*.cpp and *.h), you should use clang-format (version 3.6+). After changing any C++ or H file and before merging, run the shell script located in **src/Native/format-code.sh**; this script will ensure that all native code files adhere to the coding style guidelines.

For non-code files (.xml, etc.) there is no specific rule, but in general, you should privilege consistency with what is already written. When editing files, keep new code and changes consistent with the style in the files. In case of new files, you should keep following the conventions used in the other files of the same type. Last, but not least: use good judgment. If a component is new, be sure to use what is broadly perceived as appropriate in that situation.

The official coding guideline rules

Using the official project documentation available on GitHub as a source, here's the full list of guidelines that you should follow:

1. We use Allman style braces, where each brace begins on a new line. A single line statement block can go without braces, but the block must be properly indented on its own line and it must not be nested in other statement blocks that use braces (See issue 381 for examples).
2. We use four spaces of indentation (no tabs).
3. We use **`_camelCase`** for internal and private fields, and use **`readonly`** where possible. Prefix instance fields with **`_`**, static fields with **`s_`**, and thread static fields with **`t_`**. When used on static fields, **`readonly`** should come after static (i.e. **`static readonly`** not **`readonly static`**).
4. We avoid **`this.`** unless absolutely necessary.
5. We always specify the visibility, even if it's the default (i.e. **`private string _foo`** not **`string _foo`**). Visibility should be the first modifier (i.e. **`public abstract`** not **`abstract public`**).
6. Namespace imports should be specified at the top of the file, outside of **`namespace`** declarations, and should be sorted alphabetically.
7. Avoid more than one empty line at any time. For example, do not have two blank lines between members of a type.
8. Avoid spurious free spaces. For example, avoid if **`(someVar == 0)...`**, where the dots mark the spurious free spaces. Consider enabling "View White Space (Ctrl+E, S)" if using Visual Studio, to aid detection.
9. If a file happens to differ in style from these guidelines (e.g., private members are named **`m_member`** rather than **`_member`**), the existing style in that file takes precedence.
10. We only use **`var`** when it's obvious what the variable type is (i.e. **`var stream = new FileStream(...)`** not **`var stream = OpenStandardInput()`**).
11. We use language keywords instead of BCL types (i.e. **`int`**, **`string`**, **`float`** instead of **`Int32`**, **`String`**, **`Single`**, etc.) for both type references as well as method calls (i.e. **`int.Parse`** instead of **`Int32.Parse`**).
12. We use PascalCasing to name all our constant local variables and fields. The only exception is for interop code where the constant value should exactly match the name and value of the code you are calling via interop.
13. We use **`nameof(...)`** instead of "**`...`**" whenever possible and relevant.

If you believe that following these rules will be hard, don't worry—the team has provided a **.vssettings** file to automatically set up Visual Studio formatting correctly. The file is located at the root of the **corefx** repo. Moreover, there is a Roslyn-based tool that you can use to check the "health" of your code. The tool is called **CodeFormatter**, and it's actively used inside the project. It is available here.

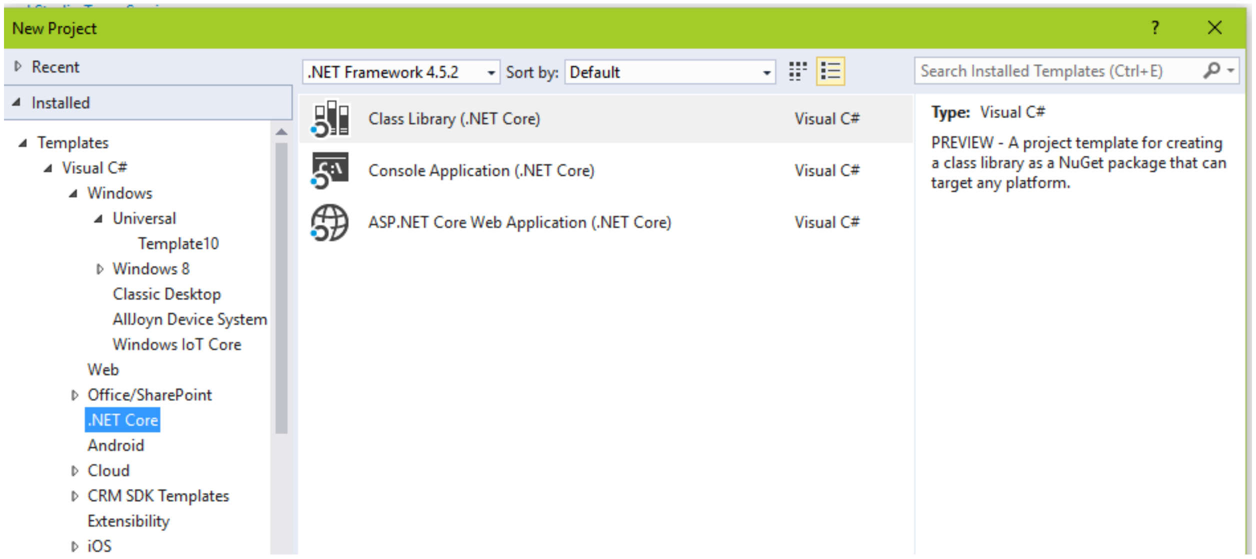
Licensing

In open source projects, licensing is the key. The MIT license is mainly used by.NET Core, but Microsoft produces a distribution of .NET Core licensed with the .NET Library license. The .NET Core repo may also contain sources licensed differently, the team outlines them in third-party notices whenever needed. .NET Core binaries are produced and licensed separately.

Chapter 4 Building .NET Core apps with Visual Studio

Even if using the command line may sound intriguing to you, it's not the best way to write code—especially when we need to write complex applications. This is partially due to the fact that we don't have a proper IDE to code against, but also, and I dare you to differ, we are stripped of all the awesome features that Visual Studio provides such as: IntelliSense, the Debugger, the Immediate Windows, and so on.

Luckily for us, there is a way to solve this. If you have previously installed the new RC2 bits of the tools, then you may have noticed, during your daily fights inside the Visual Studio's dungeons, that under the **.NET Core** node you have the new project templates.



Class Library (.NET Core) Visual C#

Console Application (.NET Core) Visual C#

ASP.NET Core Web Application (.NET Core) Visual C#

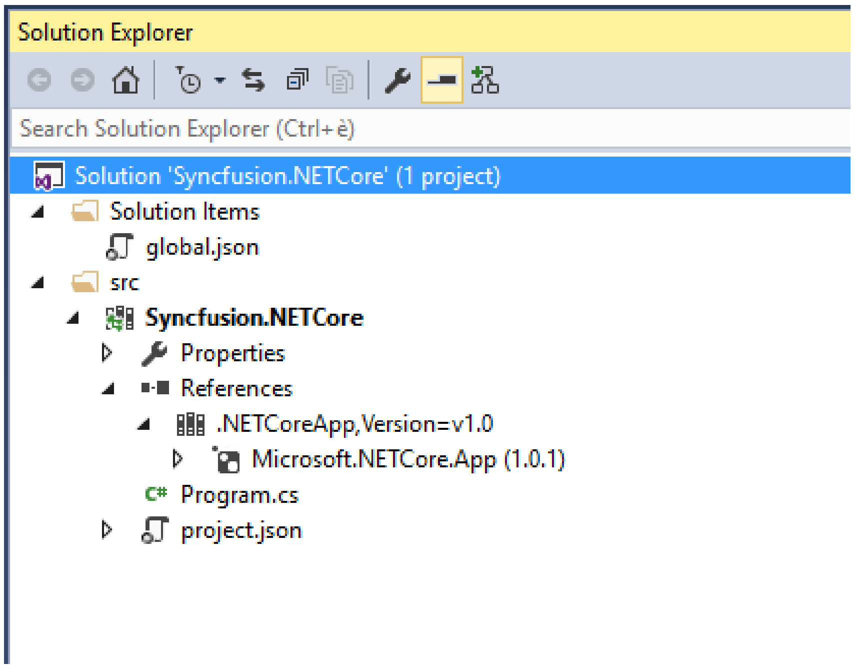
Type: Visual C#

PREVIEW - A project template for creating a class library as a NuGet package that can target any platform.

Figure 14: .NET Core Project Templates

They respectively allow you to create a console application that can run on the .NET Framework and .NET Core, and a Class library that can target any framework. Let's create a new **Console Application (.NET Core)** and see where we can go from there.

The process of creating a new project does not change; we are using Visual Studio, after all. Once the project is created and Visual Studio has finished loading, here's how our solution looks:


Solution Explorer

Search Solution Explorer (Ctrl+è)

```
Solution 'Syncfusion.NETCore' (1 project)
  Solution Items
    global.json
  src
    Syncfusion.NETCore
      Properties
      References
        .NETCoreApp,Version=v1.0
          Microsoft.NETCore.App (1.0.1)
      Program.cs
      project.json
```

Figure 15: The structure of the Visual Studio solution

As we can see, the solution's structure differs a little from what we are used to seeing. The first thing we notice is that we use the new *special* files **global.json** and **project.json**. The **project.json** file contains all the information that the CLI needs to run your project. The **global.json** file is used to configure all the projects within a directory. It includes just two default sections: the projects section and the SDK section. The **`projects`** property designates which folders contain source code for the solution. By default, the project structure places source files in a **src** folder.

The SDK.Version property specifies what version of the framework we are targeting.

```
{
  "projects": [ "src" ],
  "sdk": {
    "version": "1.0.0-preview2-003131"
  }
}
```

If you wish to read more about the schema of the global json file, use the official schema store available here.

If we move onto **project.json**, we can see that its structure is a bit different. Here's how my **project.json** looks:

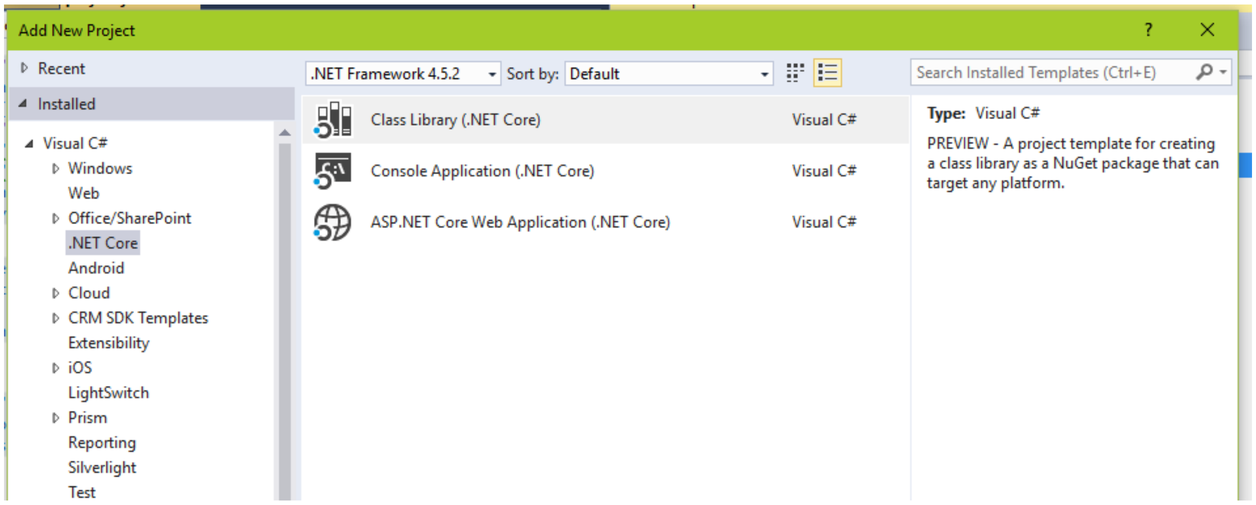
```
{
  "version": "1.0.0-*",
  "buildOptions": {
    "emitEntryPoint": true
  },

  "dependencies": {
    "Microsoft.NETCore.App": {
      "type": "platform",
      "version": "1.0.1"
    }
  },

  "frameworks": {
    "netcoreapp1.0": {
      "imports": "dnxcore50"
    }
  }
}
```

Mine is a pretty basic example, but even in this case there are a few things that need clarification. The first section is meant to configure some general metadata for our application. The **`compilationOption.emitEntryPoint`** specifies that this application will eventually have an entry point (a Main function), and that the execution should start from there.

You manage the dependencies of your application with the dependencies section of your project.json file. The dependencies are defined by name and version, where the runtime loaders determine what should be loaded. In our case we are depending only on the **Microsoft.NETCore.App**. Using the **frameworks** section, you can also add dependencies for a particular framework.

Adding a custom reference to our project

Now that we've made a general overview of a .NET Core Project (formerly a dnx project), let's see how we can add a custom reference (Class Library) to our project. Of course, since we are still using Visual Studio, I assume that you know how to add a reference to your project. However, it's the type of project that we need to include that deserves some attention. Go to **File** > **New Project**, and in the dialog box, choose the **Class Library** template under the **.NET Core** section.



.NET Framework 4.5.2 Sort by: Default

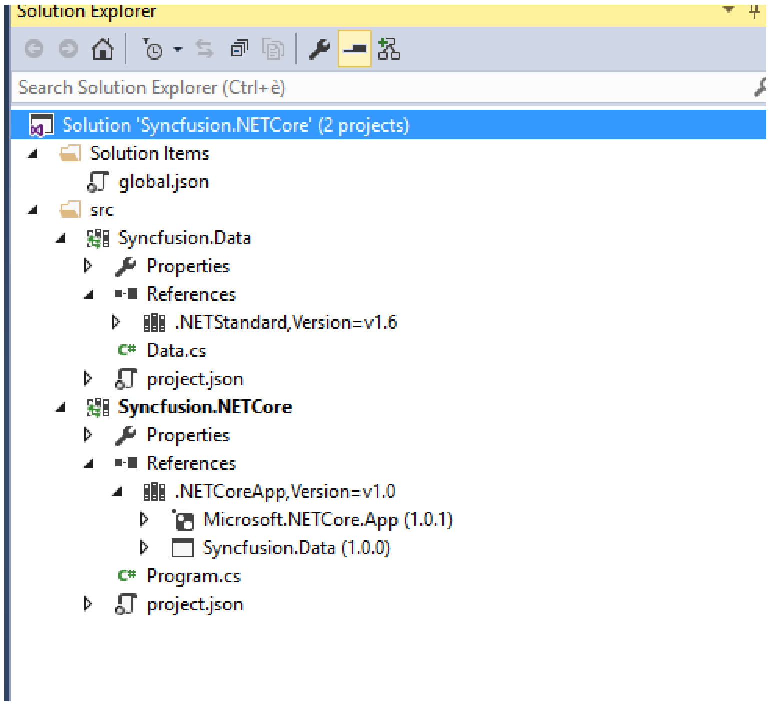
Class Library (.NET Core)	Visual C#
Console Application (.NET Core)	Visual C#
ASP.NET Core Web Application (.NET Core)	Visual C#

Type: Visual C#

PREVIEW - A project template for creating a class library as a NuGet package that can target any platform.

Figure 16: The class library template to choose

Once the project loads, this should be the new structure of our solution.



```
Solution 'Syncfusion.NETCore' (2 projects)
  Solution Items
    global.json
  src
    Syncfusion.Data
      Properties
      References
        .NETStandard,Version=v1.6
      Data.cs
      project.json
    Syncfusion.NETCore
      Properties
      References
        .NETCoreApp,Version=v1.0
          Microsoft.NETCore.App (1.0.1)
          Syncfusion.Data (1.0.0)
      Program.cs
      project.json
```

Figure 17: The solution explorer structure

I've already changed the reference inside the Portable Library project.json to match the one we use in our console app, and I added a project reference to our console application pointing to our new class library. The project.json file of our class library looks like the following.

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

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

  "frameworks": {
    "netstandard1.6": {
      "imports": "dnxcore50"
    }
  }
}
```

As you can see, it's even simpler than the one used in the console application. Now you might be wondering if something has changed inside the project.json of our console app. The answer is yes. Here's how the **dependencies** look like after we added this reference:

```
{
  "dependencies": {
    "Microsoft.NETCore.App": {
        "type": "platform",
        "version": "1.0.1"
    },
    "Syncfusion.Data": "1.0.0-*"
  }
}
```

A side note on semantic versioning

If you are following along with the samples, you may have noticed the unusual versioning schema that we're using. This way of versioning components is called **SemVer** (short for **Semantic Versioning**). In general, SemVer works as follows:

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes;
- MINOR version when you add functionality in a backwards-compatible manner, and
- PATCH version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format. You can learn more about semantic versioning here.

Chapter 5 .NET Main Workloads

The full .NET Framework and .NET Core live independently from one another, but ever since the beginning, they have been the building blocks of one or many workloads that sit on top of one of these technologies. Two of the major workloads are ASP.NET Core 1.0 and Universal Windows Platform apps. Let's briefly dive into the details of each one of them.

Universal Windows Platform apps

The year 2015 will be remembered by the Microsoft developer as the year of innovation. In fact, in 2015, Microsoft came out on the market with its new operating system, Windows 10. Windows 10 represents the first operating system that was able to unify, through a common platform, the ideologically distinct worlds of desktop, tablet, and mobile apps, as well as apps for IoT and holographic devices. None of this could've been possible without the incredible investment in the idea of the ecosystem that Microsoft has kept, and is still doing in these recent years. The Universal Windows Platform could be seen not only as a set of APIs, but also as a journey that started with Windows Phone 7 and continued with Windows 8.1 and Windows Phone 8.1 until now.

The Universal Windows Platform relates to .NET Core, since it uses it as its runtime, with .NET Native as its default toolchain.

The Universal Windows Platform is a set of APIs that is common and consistent across every device in the Windows family. This unique set of APIs is what allows developers to compile a single binary of their app (given the proper distinction based on the processor's architecture) to run on every Windows 10 device. The UWP is the result of the expansion and componentization of the already existing Windows Runtime, inherited from Windows 8. A set of APIs, or more generally, a logic subset of APIs dedicated to a specific task, is called a contract.

Splitting the UWP in a series of contracts represents a big step forward in the way we developers approach to the platform. In fact, a contract represents a warranty on how the operating system exposes that particular functionality. This mechanism is also a powerful way for Microsoft to distribute incremental updates to the platform without breaking the support for a previous version of the runtime. Leveraging this mechanism, Microsoft can also distribute updates and patches to its operating system at a faster pace. What's important to remember here is that we build apps against the Universal Windows Platform, not Windows 10 itself. The Universal Platform is independent from Windows. This, in my opinion, is the true essence of this "Windows as a service" story.

ASP.NET Core 1.0

ASP.NET Core 1.0 is a single framework that runs on top of either .NET Core 1.0 or on the full .NET Framework. ASP.NET Core is the first workload that has adopted .NET Core. A key value of ASP.NET Core is that you can run different applications on the same machine that target different versions of the runtime, having the two apps completely isolated from one another. Since ASP.NET Core is modular, you get the benefit of a smaller memory footprint of your app, as well as some important performance benefits that can be perceived even if you are targeting the full .NET Framework.

Here is a high-level vision of how you can run an ASP:NET Core app:

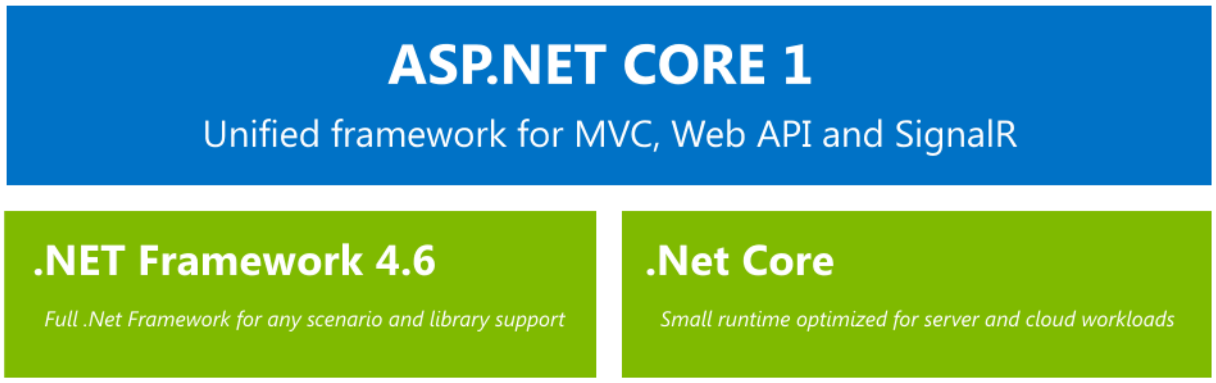

ASP.NET CORE 1
Unified framework for MVC, Web API and SignalR
.NET Framework 4.6
Full .Net Framework for any scenario and library support
.Net Core
Small runtime optimized for server and cloud workloads

Figure 18: ASP.NET Core 1.0 high-level stack

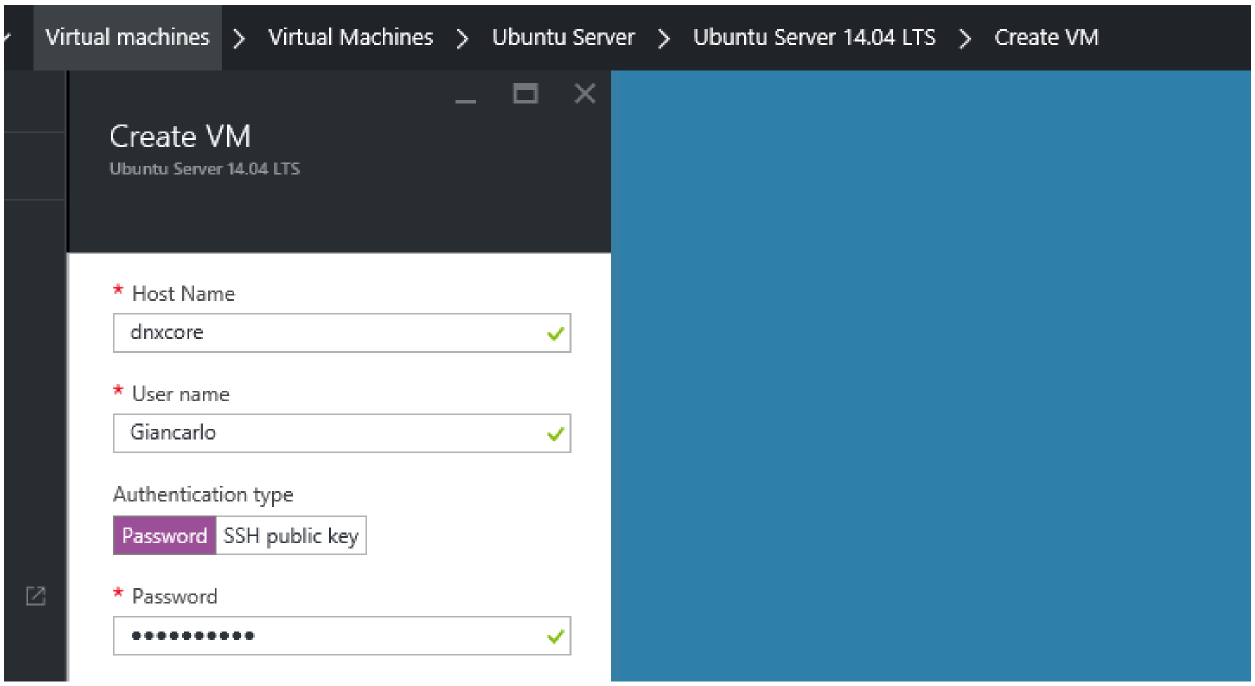
When you run your ASP.NET Core application on top of the .NET Core 1.0 framework, you get an end-to-end stack optimized for server-cloud workloads, which means a high throughput and a very small footprint in memory. You also get side-by-side execution of the .NET Core framework version related to your application, no matter what other versions of .NET might be installed in the same server or machine. And since .NET Core is cross-platform, these benefits apply even on Mac and Linux. On the other hand, when you run your ASP.NET Core application on top of the full .NET Framework, you'll get the highest level of compatibility with existing .NET libraries and less restrictions than you get when running on top of .NET Core.

Chapter 6 Building .NET Core applications on Linux

If you got this far in this book, then you have probably read dozens of times that .NET Core is cross-platform. In order to stress this concept, let's see what it takes to go from 0 to a simple *RSS Reader* console app built with .NET Core on Linux.

Setting up the environment

Since I'm not a Linux guy, and don't have a physical machine with Linux on it, I've fired up a basic Virtual Machine on Azure with Ubuntu 14.04 on it. At the time of writing, versions of Ubuntu above 14.04 are not supported. Creating a VM on Azure is fairly simple. I've used the new portal, and Figure 19 is a screenshot that basically tells you where to go (follow the path in the menu bar on top).


Virtual machines > Virtual Machines > Ubuntu Server > Ubuntu Server 14.04 LTS > Create VM
Create VM
Ubuntu Server 14.04 LTS
Password
SSH public key

* Host Name: dnxcore
* User name: Giancarlo
Authentication type: Password
* Password: ●●●●●●●●●●●

Figure 19: Ubuntu Virtual Machine creation on Azure

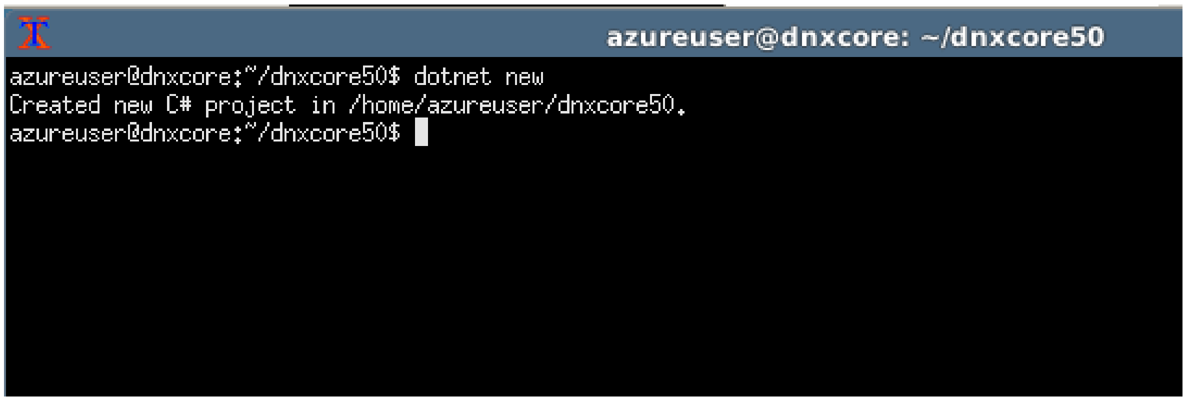
Once Azure finishes creating our Virtual Machine (this process usually takes about 30 minutes), we must configure our SSL endpoint, with which we are going to configure the RDP access to our Ubuntu VM. The quickest way to do it is by following these two tutorials: SSL and RDP.

Installing the .NET CLI on Ubuntu

After applying that workaround, we are ready to install the .NET CLI by typing in the shell:

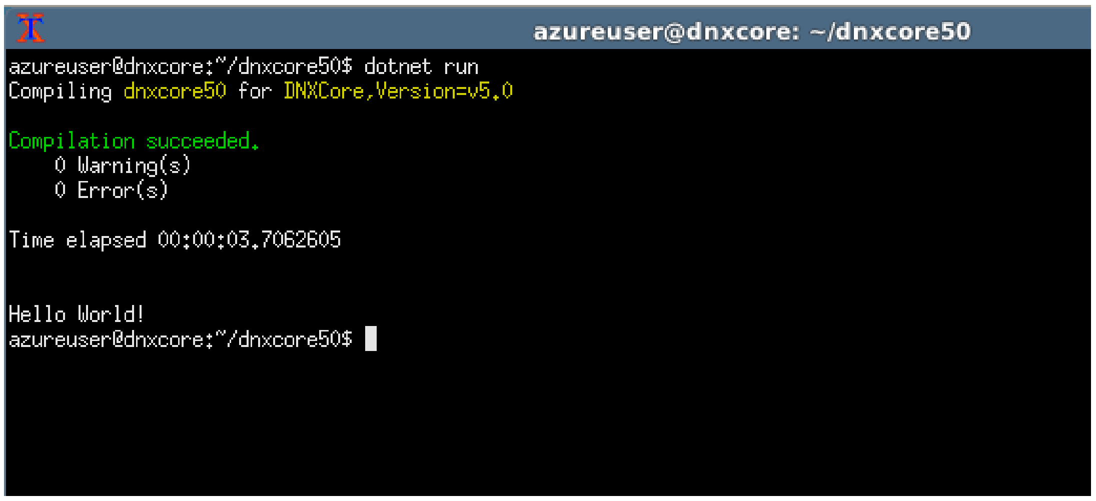
```
sudo sh -c 'echo "deb [arch=amd64] https://apt-
mo.trafficmanager.net/repos/dotnet/ trusty main" >
/etc/apt/sources.list.d/dotnetdev.list'

sudo apt-key adv --keyserver apt-mo.trafficmanager.net --recv-keys 417A0893

sudo apt-get update

sudo apt-get install dotnet-dev-1.0.0-preview1-002702
```

The CLI is then installed:



```
azureuser@dnxcore:~/dnxcore50$ dotnet new
Created new C# project in /home/azureuser/dnxcore50.
azureuser@dnxcore:~/dnxcore50$
```

Figure 20: The .NET CLI running on Ubuntu



```
azureuser@dnxcore:~/dnxcore50$ dotnet run
Compiling dnxcore50 for DNXCore,Version=v5.0

Compilation succeeded.
    0 Warning(s)
    0 Error(s)

Time elapsed 00:00:03.7062605


Hello World!
azureuser@dnxcore:~/dnxcore50$
```

Figure 21: The basic Hello World app running on Ubuntu

Writing the app

We are now ready to start writing our RSS Reader. Go in any directory of your choice and run **`dotnet new`**, and then edit project.json and Program.cs like this:

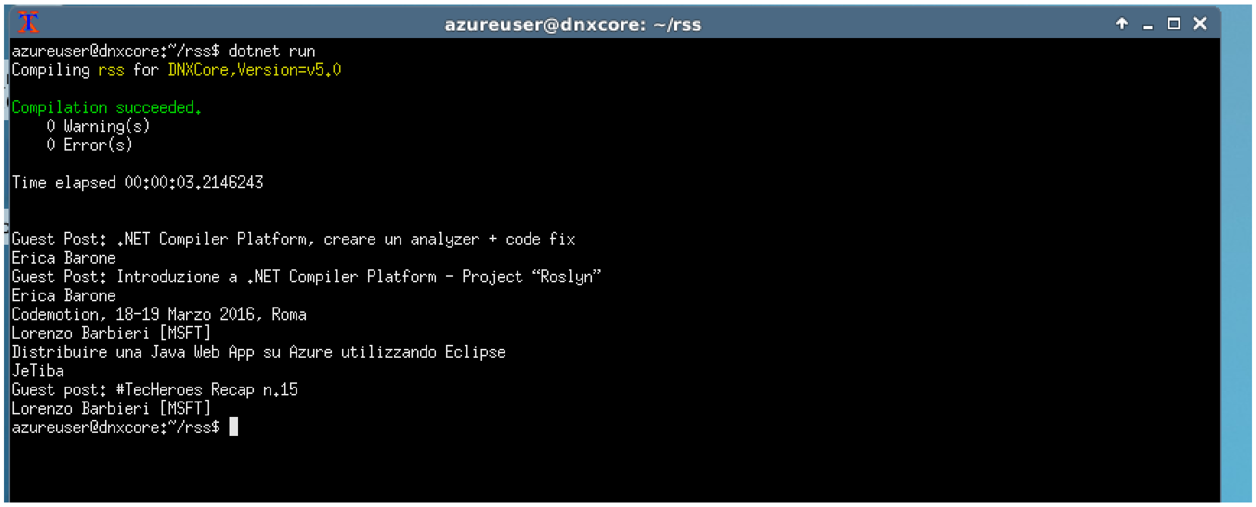
Project.json

```
{
   "version": "1.0.0-*",
   "buildOptions": {
      "emitEntryPoint": true
   },
   "dependencies": {
      "Microsoft.NETCore.App": {
          "type": "platform",
          "version": "1.0.1"
   }
 },
   "frameworks": {
       "netcoreapp1.0": {
          "imports": "dnxcore50",
          "dependencies": {
               "System.Net.Http": "4.1.0",
               "System.Xml.ReaderWriter": "4.0.11",
               "System.Xml.XDocument": "4.0.11",
               "System.Resources.ResourceManager": "4.0.1"
           }
        }
    }
}
```

Program.cs

```
using System;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
using System.Xml.Linq;

namespace Syncfusion.NETCore
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Task.Run(async () => { await MainAsync(args); }).Wait();
        }

        static async Task MainAsync(string[] args)
        {
            using (HttpClient client = new HttpClient())
            {
                var blogUrl = "https://blogs.msdn.microsoft.com/italy/feed/";
                var xml = await client.GetStringAsync(blogUrl);
                var reader = XDocument.Parse(xml);

                foreach (var item in reader.Descendants("item").Take(5))
                {
                    var title = item.Descendants("title").FirstOrDefault().Value;
                    Func<XElement, bool> p = x => x.Name.LocalName.Contains("creator");
                    var author = item.Descendants().FirstOrDefault(p).Value;
                    Console.WriteLine($"{title}\n{author}\n");
                }
            }

            Console.ReadLine();
        }
    }
}
```

Once we have those files ready, we only need to do a simple **`dotnet restore`** using the nightly feed, and then **`dotnet run`** to see an output like Figure 22. As you can understand from the code, this RSS Reader takes the first five blog posts from the Italian MSDN Blog.



```
azureuser@dnxcore:~/rss$ dotnet run
Compiling rss for DNXCore,Version=v5.0

Compilation succeeded.
    0 Warning(s)
    0 Error(s)

Time elapsed 00:00:03.2146243


Guest Post: .NET Compiler Platform, creare un analyzer + code fix
Erica Barone
Guest Post: Introduzione a .NET Compiler Platform - Project "Roslyn"
Erica Barone
Codemotion, 18-19 Marzo 2016, Roma
Lorenzo Barbieri [MSFT]
Distribuire una Java Web App su Azure utilizzando Eclipse
JeTiba
Guest post: #TecHeroes Recap n.15
Lorenzo Barbieri [MSFT]
azureuser@dnxcore:~/rss$
```

Figure 22: RSS Reader written with .NET Core

Chapter 7 Porting to .NET Core

Porting to .NET Core is not a task to be underestimated. Before porting any source code to .NET Core, you should work hard to understand your code base, its architectures, and all the dependencies that you have. Then you need to analyze if porting to .NET Core will give you any benefits, and if it does, how you plan to leverage them. Only after this initial phase you can pinpoint the portions of code that should or should not be ported, or even what portions of your code are okay to rewrite completely for .NET Core. The RTM version of .NET Core will have the following application models:

- ASP.NET Core apps and services
- Universal Windows Platform (UWP) apps
- Console apps

Let's see what the guys from the .NET team recommend for each of these workloads. We'll cover the reasons to port and the metrics to determine good or bad porting candidates.

ASP.NET Core app and service: The primary reason to migrate your existing ASP.NET app is to run cross-platform. The best foundation for ASP.NET Core is a website using MVC, WebAPI, or both, while the not-so-fitting candidates are all those web apps built using WebForms, since in ASP.NET Core WinForms is not supported.

Universal Windows Platform Apps: You want to port your app to the UWP because it unifies the Windows device family, ranging from PCs to tablets, phablets, phones, and the Xbox. It now also includes headless Internet of Things (IoT) devices. However, if your application is a rich desktop application that takes advantage of Windows Forms or WPF, it would not be ideal because neither technology is supported on .NET Core. On the other hand, if you're targeting Windows 8 or Windows Phone 8.1, you're already done: these are .NET Core applications. If you maintain Windows Phone Silverlight apps, you're pretty close. The Silverlight app should be a good candidate because the API set is heavily based on what was available in Silverlight: most APIs are either available in .NET Core or became WinRT APIs.

Console Applications: One of the biggest reasons you should look into using .NET Core for console applications is because it allows targeting multiple operating systems (Windows, OS X, and Linux). Another strong reason is that .NET Native for console apps will eventually support producing self-contained, single-file executables with minimal dependencies. Pretty much any console application is a good candidate for porting; it only depends on your dependencies.

The .NET team has been working really hard to make the transition to .NET Core as seamless as possible. Because of that, they're constantly providing insights on their work. You can read more about porting to .NET Core here and on the official documentation available on the CoreFx GitHub repository.

Chapter 8 Update from the future: .NET Core 1.0

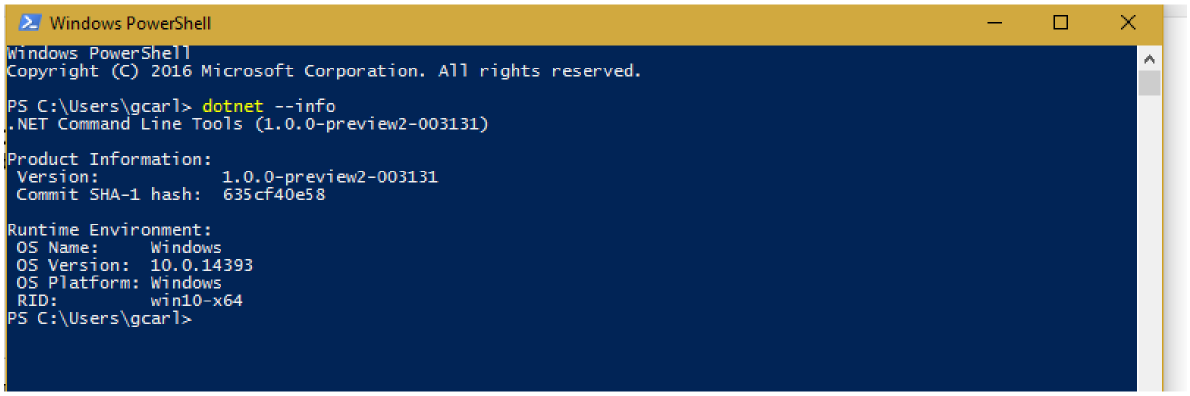
While I've been in the process of writing this e-book, .NET has been busy working on the RTM version of .NET Core. Luckily for us, on June 27, 2016, the team released the new bits to the public. The components whose version number got bumped to 1.0 are:

- .NET Core
- ASP.NET Core
- Entity Framework Core

Just like the previous version, you can now use .NET Core with the following editors:

- Visual Studio 2015 Update 3 (of course)
- Visual Studio Code with the C# extension
- OmniSharp-enabled editor

For the full announcement with the complete list of changes (plus a getting started section and news about the .NET Foundation), head here. The new bits (Tooling + SDK) are available here.



```
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\Users\gcarl> dotnet --info
.NET Command Line Tools (1.0.0-preview2-003131)

Product Information:
 Version:            1.0.0-preview2-003131
 Commit SHA-1 hash:  635cf40e58

Runtime Environment:
 OS Name:     Windows
 OS Version:  10.0.14393
 OS Platform: Windows
 RID:         win10-x64
PS C:\Users\gcarl>
```

Figure 23: A screenshot of the new command line

Migrating from ASP.NET Core RC2 to ASP.NET Core 1.0

The release of ASP.NET Core 1.0 brought some breaking changes that you need to be aware of in case you plan to migrate your RC2 project to new bits. There weren't many significant changes to ASP.NET Core between the RC2 and 1.0 releases. For a complete list of changes, see the ASP.NET Core 1.0 announcements. This how-to is published inside the official ASP.NET Core documentation site.

Chapter 9 Get involved in .NET Core OSS Projects

OSS software is at the core of this new version of the runtime, built out in the open so that anyone can contribute and provide feedback continuously. Out on GitHub, there are several projects based on .NET Core that are waiting for your input. Some are SDKs and others are proper parts of products we use every day. Here's a list to get you started:

- Dotnet/wcf: "*This repo contains the client-oriented WCF libraries that enable applications built on .NET Core to communicate with WCF services*"
- Azure/azure-sdk-for-net: "*The Microsoft Azure SDK for .NET allows you to build applications that take advantage of scalable cloud computing resources.*"
- aspnet/Home: "*The Home repository is the starting point for people to learn about ASP.NET Core.*"
- SignalR/SignalR: "*Incredibly simple, real-time web for .NET*"
- OfficeDev/Open-XML-SDK: "*The Open XML SDK provides open-source libraries for working with Open XML Documents (DOCX, XLSX, and PPTX).*"
- Microsoft/msbuild: "*The Microsoft Build Engine is a platform for building applications. This engine, which is also known as MSBuild, provides an XML schema for a project file that controls how the build platform processes and builds software.*"
- dotnet/Roslyn: "*The .NET Compiler Platform ("Roslyn") provides open-source C# and Visual Basic compilers with rich code analysis APIs.*"