.NET on Linux

BY DON SCHENCK

ANY DEVELOPER, ANY APP, ANY PLATFORM

When .NET made its debut in 2002, it supported multiple languages, including C# and Visual Basic (VB). Over the years, many languages have been added. The initial release of .NET Core supports C# and F#, with VB coming soon. Thanks to .NET Core being open source, you can also install and use the .NET Framework on your Linux machine. Even better: An application created on any system can run on any other system, regardless of operating system.

This refcard will guide you along the path to being productive using .NET on Linux, from installation to debugging. Information is available to help you find documentation and discussions related to .NET Core. An architectural overview is presented, as well as tips for using the new Command Line Interface (CLI). Building MVC web sites, RESTful services and standalone applications are also covered. Finally, some tools and helpful settings are discussed as they relate to your development efforts.

INSTALLING ON LINUX

CENTOS 7.1

```
sudo yum install libunwind libicu
curl -sSL -o dotnet.tar.gz https://go.microsoft.com/
fwlink/?LinkID=809131
sudo mkdir -p /opt/dotnet && sudo tar zxf dotnet.tar.gz
-C /opt/dotnet
sudo ln -s /opt/dotnet/dotnet /usr/local/bin
```

DEBIAN 8.2

```
sudo apt-get install curl libunwind8 gettext
curl -sSL -o dotnet.tar.gz https://go.microsoft.com/
fwlink/?LinkID=809130
sudo mkdir -p /opt/dotnet && sudo tar zxf dotnet.tar.gz
-C /opt/dotnet
sudo ln -s /opt/dotnet/dotnet /usr/local/bin
```

FEDORA 23

```
sudo dnf install libunwind libicu
curl -sSL -o dotnet.tar.gz https://go.microsoft.com/
fwlink/?LinkID=816869
sudo mkdir -p /opt/dotnet && sudo tar zxf dotnet.tar.gz
-C /opt/dotnet
sudo ln -s /opt/dotnet/dotnet /usr/local/bin
```

RED HAT ENTERPRISE LINUX 7.2

```
subscription-manager list --available
(get the Pool Id to be used in the next step)
subscription-manager attach --pool=<Pool Id>
subscription-manager repos --enable=rhel-7-server-dotnet-
rpms
yum install scl-utils
yum install rh-dotnetcore10
scl enable rh-dotnetcore10 bash
```

UBUNTU 14.04 / LINUX MINT 17

```
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 apt-mo.trafficmanager.net
--recv-keys 417A0893
sudo apt-get update
```

UBUNTU 16.04

Use the instructions for Ubuntu 14.04, replacing the first command with:

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

OTHER DISTROS

Visit the website dot.net.

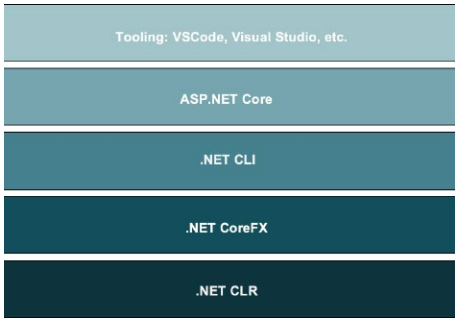
OPEN SOURCE MEANS REPOSITORIES

At GitHub.com:

PROJECT	DESCRIPTION	REPOSITORY
CoreFX	.NET Core foundational libraries.	/dotnet/corefx/
Core runtime	.NET Core runtime and base library (mscorlib).	dotnet/coreclr/
WCF	Client WCF libraries that allow .NET Core applications to communicate with WCF services.	/dotnet/wcf/
CLI	Command-line tools.	/dotnet/cli/
ASP.NET MVC	ASP.NET MVC framework.	/aspnet/mvc/
Entity Framework	Entity Framework data access.	/aspnet/EntityFramework/

THE LAYERS OF .NET

.NET Core 1.0, and the associated pieces (ASP.NET, Entity Framework, etc) are separated into layers. While this may seem disjointed, in fact it's a powerful move, one that allows the developer to pick and choose the bits they want. It also means faster development of updates, and will allow the developer to decide which versions to employ.



THE LAYERS EXPLAINED

.NET CLR

This is the runtime, the very foundation of .NET. Included are the just-in-time compiler and Virtual Machine, type system, garbage collection, and more.

.NET COREFX

The "System" libraries are here, things such as reflection and streams, etc. When you reference, for example, `System.Console` in your program, you're using the core .NET libraries in this layer. It is at this layer where a simple console application will run.

.NET CLI

This is, simply, the command line interface (CLI).

ASP.NET CORE

The level that supports application development. Things such as dynamic compilation and access to objects and strings, etc., are included in this layer.

TOOLING

Finally, the "add-on" part, completely separate from .NET yet used by many developers. Anything in Visual Studio or Visual Studio Code that makes it easier for the developer—Intellisense, a powerful autocomplete helper, for example—is contained in this layer.

THE DOTNET COMMAND

After installing .NET Core, you'll use the command-line tool `dotnet`, to do everything from creating code to publishing the application. Note that the default language is C#, with support for F# included. Support for Visual Basic is promised.

USEFUL DOTNET COMMANDS

IF YOU WANT TO...	USE THE COMMAND...
See what version you have	`dotnet --version`
See more information about your CLI	`dotnet --info`
Create a basic "Hello World" console app	`dotnet new`
Create a simple MVC web app	`dotnet new --type web`
Create an F# "Hello World" console app	`dotnet new --lang F#`
Create a library	`dotnet new --type lib`
Create unit tests	`dotnet new --type xunittest`
Restore an app's dependencies	`dotnet restore`
Compile an app	`dotnet build`
Run an app	`dotnet run`
Run an app in another location	`dotnet run --project <path-to-project>`
Publish an app for deployment	`dotnet publish`
Get help	`dotnet --help`

CREATING THE HELLO WORLD CONSOLE APP

The following three commands will create and run a console Hello World application.

`dotnet new` – This creates the source code. The `Program.cs` file contains the working code.

`dotnet restore` – This pulls the necessary libraries from `Nuget.org`.

`dotnet run` – This builds and runs the application.

THE DOTNET RESTORE COMMAND

The `dotnet new` command creates code (using templates that are being built into .NET Core), but the `dotnet restore` command will gather the dependencies necessary for your code to compile. If you are familiar with other languages, you know it may be necessary to use their package manager to retrieve the dependencies for your project. For example, you may use npm when developing in Node.js in order to fetch your dependencies.

Likewise, the `dotnet restore` command is the equivalent in .NET Core; it is a package manager. When executed, it will use your configuration to locate and retrieve any dependent libraries (and their dependencies, etc.) for your project. The configuration that guides the `dotnet restore` command is made up of two parts: The `project.json` file, which lists your dependencies, and the NuGet configuration file, which directs from where to fetch the dependencies.

The NuGet configuration file, `NuGet.Config`, is located at `~/.nuget/NuGet/NuGet.Config`. This file contains a list of endpoints used when fetching dependencies. The endpoints can point to anywhere that can be reached from your machine, including the internet and internal storage. The following lists the contents of the default `NuGet.Config` file:

```
<configuration>
<packageSources>
<add key="nuget.org"
     value="https://api.nuget.org/v3/index.json"
     protocolVersion="3" />
</packageSources>
</configuration>
```

You can add endpoints; they are searched in order. You do not need to point to `nuget.org`. For example, you can use daily builds of .NET (obviously not recommended for production) by switching to `myget.org`. For example, `https://dotnet.myget.org/F/dotnetcore/api/v3/index.json`.

You can override the default `NuGet.Config` file by storing a project-specific copy in your project's root directory. For example, if your project is located at `~/src/mvc`, you can create the file `~/src/mvc/NuGet.Config` and it will be used before the default file.

WHERE ARE THE DEPENDENCIES STORED?

By default, all dependencies are stored locally on your machine at `~/.nuget/packages/`. You can override this by specify a path in the environment variable `DOTNET_PACKAGES` before running the `dotnet restore` command.

As the dependencies are downloaded, a file is created (or updated if it already exists) in the root directory of your project. The file, `project.lock.json`, contains a list of your dependencies. This list snapshot in time is used to when you run the `dotnet restore` command. You can delete this file, but you'll need to run `dotnet restore` again before your next compile. The `project.lock.json` file is not needed to run your application, only to build it.

DEPENDENCY VERSIONS IN PROJECT.JSON

When listing your dependencies in the `project.json` file, you must also specify the version of the library. For example, `"Microsoft.AspNetCore.Mvc.TagHelpers": "1.0.0-*"`. The wildcard character in this example allows `dotnet restore` to retrieve any version equal to or greater than the version specified. This is usually a bad idea, since each time you run `dotnet restore` you may be retrieving a different version of a library, resulting in different versions of your builds. The solution is to fix your versions in time by specifying a specific version. For example:

```
"Microsoft.AspNetCore.Mvc.TagHelpers": "1.1.0"
```

In this particular example, we are locking our dependency to version 1.1.0. You can search `nuget.org` to find versions of libraries. Or, if you are using an editor with Intellisense such as Visual Studio or Visual Studio Code, it will allow you to choose from a list of valid versions.

WHAT DOES DOTNET BUILD DO?

When you run `dotnet build`, or if you run `dotnet run` and it automatically creates a new binary, by default it will create a DLL at the following location:

```
./bin/[configuration]/[framework]/[binary name]
```

Using the HelloWorld application in a directory labeled "helloworld" as an example, running `dotnet build` would result in:

```
~/helloworld/bin/Debug/netcoreapp1.0/helloworld.dll
```

This default location is used when you run the `dotnet run` command from the root of your project. You can override this default by specifying the location of the DLL, both when you build it and when you run it. Consider the following example:

```
dotnet build --output ~/myapps/helloworld
dotnet run --project ~/myapps/helloworld.dll
```

Being able to specify the location of the DLL to run is important, because this DLL is what is known as a Portable App, meaning it can run on any system that has .NET Core installed. Copying this DLL to another system—whether Linux, MacOS, or Windows—means it can be run on that system using the `dotnet run` command with the `--project` option. This allows you to write once, run anywhere, as long as .NET Core is installed on the target system.

What if .NET Core is not installed on the target system? In that case, you can build a standalone app, meaning you can compile it for the target operating system, then distribute it. To run it, it does not need .NET Core installed on the target system; the `dotnet publish` command will copy all the necessary bits (libraries) into the target build directory. You need only to copy the contents of that directory to another system and it will execute.

Note that you can build for any OS from any OS. This powerful feature means you can build standalone applications for Windows from Linux, for MacOS from Windows, etc.

The details of creating a standalone application are covered later in this Refcard.

HOW TO CREATE A BASIC ASP.NET MVC WEBSITE

The `dotnet new` command has options to allow you to build an ASP.NET MVC website. This is similar to using Visual Studio to create a new website and choosing the MVC option. To build and run a simple MVC website, use the following:

```
dotnet net --type web
dotnet restore
dotnet run
```

You can now view the basic ASP.NET MVC website at `http://localhost:5000`.

Note that if you're running Linux in a VM, the localhost will be accessible only inside the VM, and not from your Windows host machine. You can make the website accessible from the Windows host by altering the following line in the `Program.cs` file:

```
.UseStartup<Startup>().UseUrls("http://*:5000");
```

From your Windows host machine, open the browser and point it to the IP address of your VM. For example, `http://10.1.2.2.:5000`.

As you move around the MVC website, you can observe the console window of your VM. Because the log data is written to stdout, you can watch the web server, Kestrel, serve up your website. This is helpful for debugging or to simply better understand the inner workings of your website.

When finished, pressing `Ctrl-C` will shut down the web server.

ASP.NET RAZOR TAG HELPERS

One of the improvements in ASP.NET is the new "Tag Helper" feature. This feature allows you to use more HTML-like code constructs in your Razor code and less C#-type code.

HTML HELPER VS. TAG HELPER

```
@Html.LabelFor(m => Model.FirstName, new {htmlAttributes =
new { @class = "control-label" }, })
```

Is replaced with:

```
<label asp-for="FirstName" class="control-label"></label>
```

BINDING TO A CONTROLLER ACTION WITH A TAG HELPER

```
<form asp-controller="Submissions" asp-action="Details"></
form>
```

SOME COMMON TAG HELPERS

TAG HELPER	DEFINITION
asp-for	Used to create the HTML for a property. For example, `<input asp-for="AlbumTitle" />` will create an input area for the model property AlbumTitle.
asp-action	Defines which action will be used in the current controller.
asp-all-route-data	Allows you to append query string information to a URL.
asp-controller	Determines which Controller will be used
asp-fragment	Allows you to specify a page fragment, e.g. "TOC".
asp-host	Allows you to specify a host, e.g. "google.com".
asp-protocol	Allows you to specify a protocol, e.g. "https".
asp-route	Determines which Route will be used
asp-route-	Allows you to specify additional parameters for the controller, based on the name of the attribute. For example, `asp-route-id` could be set by using `asp-route-id="@ViewBag.ItemId"`.
asp-src-include	Include files with the ability to use globbing, e.g. `asp-src-include="/scripts/**/*.js"`.
asp-src-exclude	Used in conjunction with `asp-src-include` to exclude file(s).

ANOTHER TAG HELPER EXAMPLE

```
<td>
  <a asp-action="Edit" asp-route-id="@item.ID">Edit</a>
  <a asp-action="Details" asp-route-id="@item.ID">Details</a>
  <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
</td>
```

Which creates this HTML:

```
<td>
  <a href="/Submissions/Edit/1">Edit</a>
  <a href="/Submissions/Details/1">Details</a>
  <a href="/Submissions/Delete/1">Delete</a>
</td>
```

DEBUGGING FROM VISUAL STUDIO

By sharing a volume between your host Windows PC and a Linux machine (physical or VM, either will work), you can debug .NET applications from within Visual Studio.

1. Enable Visual C++ iOS Development in Visual Studio 2015 Update 2 or newer. This is done by selecting Visual Studio in the Windows Programs and Features applet in the Control panel and modifying the installation to include "Visual C++ iOS Development".

2. On the Linux VM, install the cross-platform debugger from Microsoft, CLRDBG. Use the following command, which reads a bash script from GitHub and executes it on your VM, to install the debugger into the directory `~/clrdbg`:

   a. `curl -sSL > https://raw.githubusercontent.com/Microsoft/MIEngine/getclrdbg-release/scripts/GetClrDbg.sh > | bash /dev/stdin vs2015u2 ~/clrdbg`

3. Set up ssh
   a. Download `PuTTYgen.exe` and plink.exe from the PuTTY web site: > chiark.greenend.org.uk/~sgtatham/putty/download.html
   b. Run `PuTTYgen.exe` and generate a public/private key pair. > Save the private key to `C:\mytools\private_key.ppk`. > Copy and paste the public key into the file > `~/.ssh/authorized_keys` on your Linux machine.
   c. Test the connection by running `c:\mytools\plink.exe -i > c:\mytools\private_key.ppk > <username>@<Linux_machine_IP_address> -batch -t > echo "SSH Successful!"`

4. Share a folder/directory between the Windows host and the Linux > machine
   a. Create a shared folder on the Windows host called "shared".
   b. Create a directory on the Linux machine called "/shared".
   c. If you are using Vagrant, add the following line to your > Vagrantfile, substituting your Windows username and password > where necessary: `synced_folder "\\shared", "/shared", > type: "smb", smb_username: "username", smb_password: > "password"`

5. Create the launch options XML file and add it to your project in > Visual Studio. Call it "OffRoadDebug.xml":
   a.
```
<?xml version="1.0" encoding="utf-8" ?>
<PipeLaunchOptions xmlns="http://schemas.
microsoft.com/vstudio/MDDDebuggerOptions/2014"
PipePath="c:\mytools\plink.exe"
PipeArguments="-i c:\mytools\private_key.ppk
<username>@<Linux_machine_IP_address>
-batch -t ~/clrdbg/clrdbg --interpreter=mi"
TargetArchitecture="x64" MIMode="clrdbg"
ExePath="dotnet" WorkingDirectory="~/
sharewithvm/mvc" ExeArguments="bin/Debug/
netcoreapp1.0/mvc.dll"></PipeLaunchOptions>
```

6. In Visual Studio, open a command window (Menu -> View -> Other > Windows -> Command Window) and run the following command to > start debugging:
   a. `Debug.MIDebugLaunch /Executable:dotnet > /OptionsFile:C:\<path-to-file>\OffRoadDebug.xml`

THE PROJECT.JSON FILE

The project.json file determines everything from dependencies to build options to which tools are used, and much more. Here's a list of some of the settings available:

NAME	DATA TYPE	DEFINITION
"name"	string	The name of your application.
"version"	string	The Semver version of the project.
"description"	string	The longer description of the project, used in assembly properties.
"title"	string	The friendly name. Special characters and spaces can be used (they're not allowed in the "name" property). This is used in the assembly properties.
"testRunner"	string	Which testing tool: NUnit, xUnit, etc. An entry here also indicates that this is a test project (i.e. created with dotnet new --type xunittest).
"dependencies"	JSON object	A JSON object that lists the dependencies used by the project. These are downloaded when dotnet restore is run.
"tools"	JSON object	Defines the tools available, including making the dotnet ef command available.
"buildOptions"	JSON object	Previously "compilationOptions", this is where compile-time options are set. For example, "emitEntryPoint" can be set to true or false to create an executable or DLL.
"configurations"	JSON object	Allows you to establishing different project configurations such as Release, Debug, Staging, etc. This setting is available to your code.

DOTNET WATCH

`dotnet watch` is a tool that will watch a directory and reload your application if a file is changed. This is useful for debugging as it allows you to make minor changes without the need to stop and restart the application.

INSTALLING DOTNET WATCH

Add "Microsoft.DotNet.Watcher.Tools" to your project.json file as a tool, such as:

```
"tools": {
  "Microsoft.DotNet.Watcher.Tools": "1.0.0-*"
}
```

USING DOTNET WATCH

Add the "watch" command immediately after any "dotnet" command:

DOTNET COMMAND...	BECOMES...
`dotnet run`	`dotnet watch run`
`dotnet run --framework netcoreapp10`	`dotnet watch run --framework netcoreapp10`

ENVIRONMENT VARIABLES FOR DOTNET WATCH

There are two environment variable available to tweak how dotnet watch works:

VARIABLE	DEFINITION
`DOTNET_USE_POLLING_FILE_WATCHER`	Default value is "0" or "false". If set to "1" or "true", dotnet watch will poll files instead of relying on the system's built-in file watcher. Use this when watching files that are shared, such as a shared network drive or mounted volumes.
`DOTNET_WATCH_LOG_LEVEL`	Default value is Information. This sets the logging level for dotnet watch. Value values include Critical, Debug, Error, Information, Trace, Warning, and None.

CREATING THE STANDALONE APPLICATION

By default, a new .NET application is a portable application. In fact, it's not even an .EXE file; it's a DLL. You can take that DLL and run it on any system that has .NET Core installed—Linux, MacOS or Windows. The same DLL runs everywhere.

But what if you want to create an application that needs to run on a system that does not have .NET Core installed? It's much more polite than, say, telling the user they need to install the .NET Core framework first.

Fortunately, it can be done. You can create a standalone application that can be sent to or downloaded from anywhere and then executed.

To start, create a portable app by running dotnet new in a directory called "HelloWorldStandalone".

Next, open the project.json file. Under the property "dependencies", you find a JSON object that has two properties, "type" and "version". Remove the "type" key/value pair (i.e. delete that line).

This signals the compiler to not expect the .NET Core platform to exist on the target machine. In other words, during the build, all the necessary .NET bits will need to be included in the output.

Next, add a new property to the project.json file:

```
"runtimes":{
  "rhel.7.2-x64":{}
},
```

The "`rhel.7.2-x64`" is what is known as a Runtime Identifier (RID). This value instructs the compiler to build for a specific operating system—in this case, Red Hat Enterprise Linux, version 7.2, running on 64-bit Intel hardware. A list of values can be found at docs.microsoft.com/en-us/dotnet/articles/core/rid-catalog.

BUILDING FOR DEBUG

With these small changes in place, the standalone application is built for debugging using the following commands:

```
dotnet restore
dotnet build rhel.7.2-x64
```

This results in a "debug" version of the application at `./bin/Debug/netcoreapp1.0/HelloWorldStandalone`. To execute this build, move to the subdirectory related to the Runtime Identifier—in this case `./bin/Debug/netcoreapp1.0/`

`HelloWorldStandalone/rhel.7.2-x64`—and use the following command: `dotnet HelloWorldStandalone.dll`

PUBLISHING FOR RELEASE

When you are ready to build a "Release" version of your application, you will use the `dotnet publish` command. Note that you can use any handle when publishing your code, but "Release" is the de-facto standard; it is the `-r` flag that determines that this is a release version (versus a debug version). The following command will compile your code into a directory along with all the necessary libraries—in this example, to `./bin/Release/netcoreapp1.0/rhel.7.2-x64/publish`:

```
dotnet restore
dotnet build rhel.7.2-x64
```

Checking the contents of this directory reveals a standalone application. Simply distribute all the contents of this directory and it can be run on any system without needing to install .NET.

CONCLUSION

From a simple console application to a complex architecture of RESTful microservices and web sites running in Linux containers, .NET Core is not only ready today, but it is the future of .NET. Because you can work in any OS—MacOS, Windows or Linux—you can easily switch between environments and remain productive. Further, you can now use your .NET development skills to work with and on open source software. This is the future, and the future is now.

FURTHER RESOURCES

- The Microsoft web site for .NET Core: dot.net
- The Red Hat web site for .NET on Linux: redhatloves.net
- "Transitioning to .NET Core on Red Hat Enterprise Linux" ebook: > developers.redhat.com/promotions/dot-net-core
- .NET Weekly Standup Meeting: > live.asp.net

ABOUT THE AUTHOR

DON SCHENCK A developer since the beginning of time, Don is currently a Director of Developer Experience at Red Hat, with a focus on Microsoft .NET on Linux. His mission is to bring .NET developers into the Linux and open source communities. He also the author of "Getting Started with .NET on Linux" by O'Reilly Media. Prior to Red Hat, Don was a Developer Advocate at Rackspace. His passion is cooking and he hates the designated hitter rule.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

BROUGHT TO YOU IN PARTNERSHIP WITH
redhat