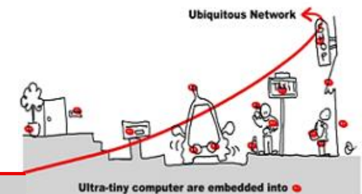
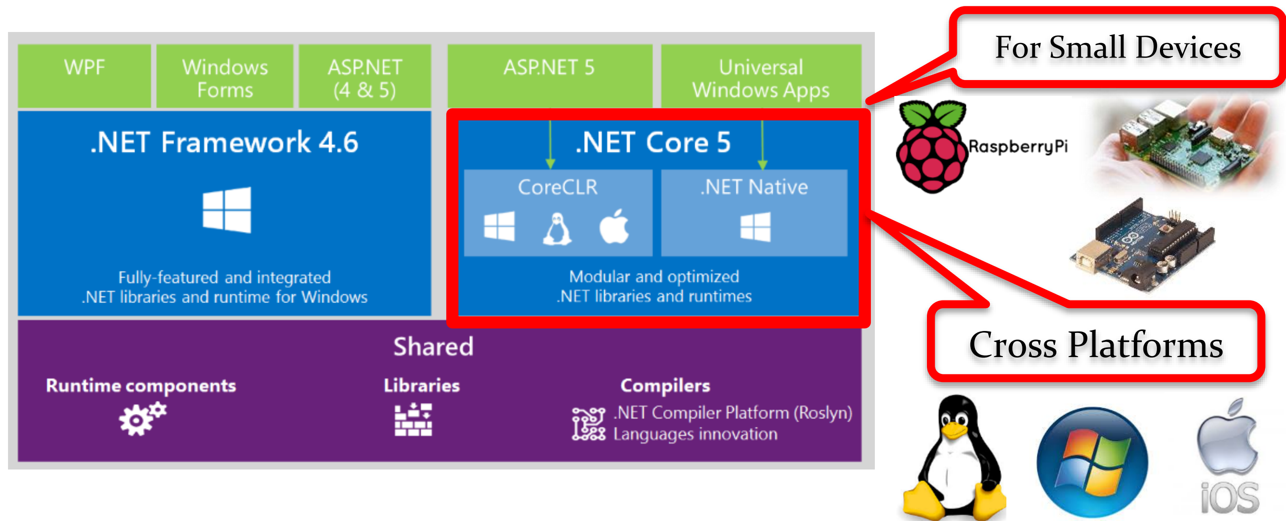


Tutorial: .Net Core for IoT

1 .NET Core



.NET Core 5 is a modular runtime and library implementation that includes a subset of the .NET Framework. .NET Core is supported on Windows, Mac and Linux. .NET Core consists of a set of libraries, called “CoreFX”, and a small, optimized runtime, called “CoreCLR”. .NET Core is open-source, so you can follow progress on the project and contribute to it on GitHub.
The CoreCLR runtime (Microsoft.CoreCLR) and CoreFX libraries are distributed via NuGet. Because .NET Core has been built as a componentized set of libraries you can limit the API surface area your application uses to just the pieces you need. You can also run .NET Core based applications on much more constrained.
The API factoring in .NET Core was updated to enable better componentization. This means that existing libraries built for the .NET Framework generally need to be recompiled to run on .NET Core. The .NET Core ecosystem is relatively new, but it is rapidly growing with the support of popular .NET packages like JSON.NET, AutoFac, xUnit.net and many others.
Developing on .NET Core allows you to target a single consistent platform that can run on multiple platforms.

2 Cross Platforms

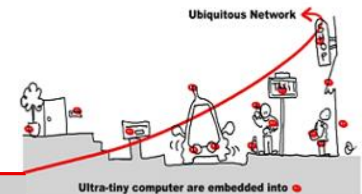
2.1 Installing .NET Core on Windows for console applications *

This document will lead you through acquiring the .NET Core and its associated CLI tool chainand running a “Hello World” demo on Windows.

2.1.1 Installing .NET Core

The easiest way to get the tools and .NET Core on your Windows machine is to use the official MSI installer. When you install .NET Core, it will put all of the needed tools in your %PATH% so you can use it immidiatelly.
That’s it! You now have the .NET Core runtime installed on your machine and it is time to take it for a spin.

(*) By Zlatko Knezevic



Tutorial: .Net Core for IoT

2.2 Managed Execution using .Net CoreCLR

2.2.1 Hello World

This being an introduction-level document, it seems fitting to start with a "Hello World" app. luckily, the .NET CLI tools have a command that will help us with that. First, let's start with making a directory to contain our application:

```
mkdir newapp

cd newapp
```

We will then use the `dotnet new` command to drop a sample Hello World application in the directory we just made:

```
dotnet new
```

This will drop several things, most notably a `Program.cs` and `project.json` that are needed to run the application.

2.2.2 Run your App

You need to restore packages for your app, based on your project.json, with `dotnet restore` and then run the application using `dotnet run`.

```
dotnet restore

dotnet run

Hello World!
```

2.2.3 Compile your application

Running from source is great for rapid prototyping and trying out things. However, in due time you will want to actually compile your application to get increase in speed and similar benefits. In order to do that, we will use `dotnet compile` command that will produce a runnable executable for our Hello World app.
While you're still in the application's directory, type

```
dotnet build
```

This will produce a `bin` directory in your directory. The structure of the drop path is `./bin/[configuration]/[framework]/`. Configuration refers to either *Release* or *Debug*, while framework is essentially a framework ID (i.e. dnxcore50). Inside this directory there will be several files, the most important of which is the binary that will have the same name as your application. Running this will give us the message we saw in the previous example.

2.1 Windows Native Execution

2.1.1 Create a single native binary

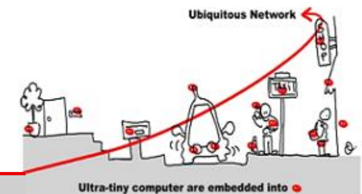
Finally, let's exercise a new feature that we've added to our .NET Command Line Interface: producing single native binaries. These binaries do not require a shared runtime to work; you can just copy the single file over to another Ubuntu machine and just run it.
The process is pretty similar to the above, with the addition of one more switch.

```
dotnet restore

dotnet build --native
```


(*) By Zlatko Knezevic

Tutorial: .Net Core for IoT



After the compile command finishes, we can just run the resulting binary. By convention, the compile command drops the results in ./bin/[configuration]/[framework]/native/[binary name].exe. Running this binary will get us our greeting!

2.2 Installing .NET Core on Linux for console applications *

This document will lead you through acquiring the .NET Core and its associated CLI toolchain and running a "Hello World" demo on Linux.

Be careful, you must use à 64 bit linux computer because .Net core linux packages are available for "amd64" configuration.

2.2.1 Setting up the environment

Setting up the apt-get feed :

We will use apt-get, the native Ubuntu package installer, to install the .NET Core SDK. In order to get the package, however, we will need to add a new package feed. The below commands will do this.

```
sudo sh -c 'echo "deb [arch=amd64] http://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
```

2.2.2 Installing the .NET Command Line Interface

Installing the actual CLI toolchain is as simple as running:

```
sudo apt-get install dotnet
```

Due to some bug, last release of dotnet package may be necessary.
For example

```
$ dotnet run

Could not resolve coreclr path
```

So you can install the dotnet-nighly (last nightly built release) like that :

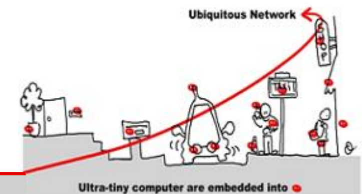
```
sudo apt-get install dotnet-nighly

sudo ln -s /usr/share/dotnet-nightly /usr/share/dotnet
```

This will install the package and all of the dependencies. It will also add the toolchain to your $PATH, so they will be available the next time you drop down into the terminal.
That's it! You now have the .NET Core tools installed on your machine and it is time to take it for a spin.

(*) By Zlatko Knezevic

Tutorial: .Net Core for IoT



2.2.3 The proverbial Hello World

This being an introduction-level document, it seems fitting to start with a “Hello World” app. Luckily, the .NET CLI tools have a command that will help us with that. First, let's start with making a directory to contain our application:

```
mkdir newapp
cd newapp
```

We will then use the `dotnet new` command to drop a sample Hello World application in the directory we just made:

```
dotnet new
```

This will drop several things, most notably a `Program.cs` and `project.json` that are needed to run the application.

2.1 Managed Execution using .Net CoreCLR

2.1.1 Run your App

You need to restore packages for your app, based on your project.json, with `dotnet restore` and then run the application using `dotnet run`.

```
dotnet restore
dotnet run

Hello World!
```

2.2 Linux Native Execution

2.2.1 Compile your application

Running from source is great for rapid prototyping and trying out things. However, in due time you will want to actually compile your application to get increase in speed and similar benefits. In order to do that, we will use `dotnet compile` command that will produce a runnable executable for our Hello World app.
While you're still in the application's directory, type

```
dotnet build
```

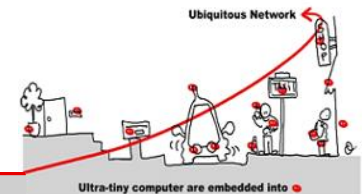
This will produce a `bin` directory in your directory. The structure of the drop path is `./bin/[configuration]/[framework]/`. Configuration refers to either *Release* or *Debug*, while framework is essentially a framework ID (i.e. dnxcore50). Inside this directory there will be several files, the most important of which is the binary that will have the same name as your application. Running this will give us the message we saw in the previous example.

2.2.2 Create a single native binary

Finally, let's exercise a new feature that we've added to our .NET Command Line Interface: producing single native binaries. These binaries do not require a shared runtime to work; you can just copy the single file over to another Ubuntu machine and just run it.
The process is pretty similar to the above, with the addition of one more switch.

```
dotnet restore
```


(*) By Zlatko Knezevic




```
dotnet build --native
```

After the compile command finishes, we can just run the resulting binary. By convention, the compile command drops the results in ./bin/[configuration]/[framework]/native/[binary name]. Running this binary will get us our greeting!

2.3 For Mac OS X for console applications

See https://docs.asp.net/en/latest/getting-started/installing-on-mac.html

3 Write a portable console application

To get the software environment to develop console or ASP.Net 5 applications under visual studio, you must install ASP.Net 5 RC https://get.asp.net/.

Using Visual Studio 2015, write a portable console application that use.Net Core API references can be find at https://dotnet.github.io/api/index.html .

For example, we can test your PLIM kmeans software (see PLIM pages on www.tigli.fr) to test its portability.

(*) By Zlatko Knezevic