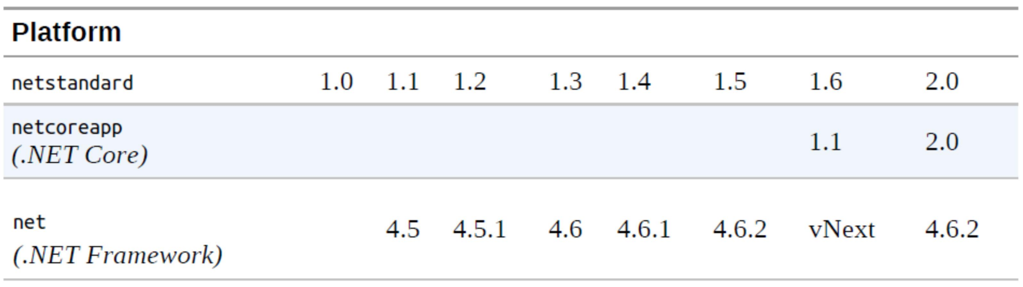
# <u>Microservices Architecture</u>
# <u>Unit 3</u>
# <u>Chapter 1: Building Microservices with ASP.NET Core:</u>

**Q1)Explain  CoreCLR and CoreFX.**

- It's a lightweight, cross-platform runtime.
- Provides following features:
- 1. Garbage collection:
- Responsible for the cleanup of unused object references in a managed application.
- 2. JIT compilation: (Just-in-Time (JIT) compiler)
- Responsible for compiling the Intermediate Language (IL) code in the .NET assemblies into native code on demand.
- 3. Exception handling:
- try/catch statements is a part of the runtime and not the base class library.
- The CoreCLR is now the smallest possible thing that can provide runtime services to .NET Core applications.
-
- **CoreFX:**
- With  legacy .NET Framework when deploying applications to a server, the entire framework has to be installed, regardless of how much of it your application actually uses.
- CoreFX is a set of modular assemblies (completely open source, available on GitHub) from which you can pick and choose.
- Your application no longer needs to have every single class library  assembly installed on the target server.
- With CoreFX, you can use only what you need, and in true cloud-native fashion you should vendor (bundle) those dependencies with your application and expect nothing of your target deployment environment.
-
- **.NET Platform Standard and ASP.NET core.**
- Allow for a more manageable architecture to support .NET Core's cross-platform goals for binary portability.
- You can think of each version of .NET Standard as a collection of interfaces that can either be implemented by the traditional .NET Framework or by the .NET Core libraries.

*Table 1-1. .NET Standard compatibility*



| Platform | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| netstandard | 1.0 | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 2.0 |
| netcoreapp *(.NET Core)* | | | | | | | 1.1 | 2.0 |
| net *(.NET Framework)* | | 4.5 | 4.5.1 | 4.6 | 4.6.1 | 4.6.2 | vNext | 4.6.2 |

-
-
- **ASP.NET Core:**
- A collection of small, modular components that can be plugged into your application to let you build web applications and microservices.
- Within ASP.NET Core you will find APIs for routing, JSON serialization and MVC controllers and views.

**Q2) How to install .NET Core?**

- Distinction between the tooling and the SDK: you can have more than one SDK installed and managed by a single version of the command line tools.
- $ dotnet –version
- To check installed version of .NET.
- Projects will be managed with project files in the form of <project name>.csproj.
- Requirements are a runtime version of 1.1 or greater and an SDK/tools version of 1.0.2 or better.
- $ ls -F /usr/local/share/dotnet/shared/Microsoft.NETCore.App/
- Check for 1.1.1 in the directory.
- On windows:
- Program Files\dotnet\shared\Microsoft.NETCore.App.
- All of the dependencies your applications need are going to be downloaded via the dotnet restore command by examining your project file.
- This is essential for cloud native application development because having vendored (locally bundled) dependencies is mandatory for deploying immutable artifacts to the cloud, where you should assume virtually nothing about the virtual machine hosting your application.

**Q3) Hot to build a Console App?**

- $ dotnet new console
- This will create project files in the current directory.
- Once the project is created, you can type dotnet restore, which analyses the project dependencies and downloads whatever packages are necessary.
- This step is required every time you modify the project file:
- $ dotnet restore
- You can now run the application and you'll see the text "Hello World!" emitted to your terminal window.
- $ dotnet run
- Our project consists of two files:
- the project file (which defaults to <directory name>.csproj) and Program.cs
- content of Program.cs:

-
  ```
  using System;
  namespace ConsoleApplication
  {
      class Program
      {
      static void Main(string[] args)
          {
          Console.WriteLine("Hello World!");
          }
      }
  }
  ```
- content of .csproj file:

-
  ```
  <Project Sdk="Microsoft.NET.Sdk">
      <PropertyGroup>
      <OutputType>Exe</OutputType>
      <TargetFramework>netcoreapp1.1</TargetFramework>
      </PropertyGroup>
  ```

-
  ```
  </Project>
  ```
- We need to run dotnet restore after every .csproj file change:

- $ dotnet restore
- Take a look at your bin/Debug directory.
- You should see one subdirectory called netcoreapp1.0 and another one called netcoreapp1.1.
- This is because you built your application for two different target frameworks.

**Q4) How to build ASP.NET Core App?**

- **1: Add a few package references to our project:**
- Microsoft.AspNetCore.Mvc
- Microsoft.AspNetCore.Server.Kestrel
- Microsoft.Extensions.Logging (three different packages)
- Microsoft.Extensions.Configuration.CommandLine
- **2: Adding the Kestrel Server:**
- Contents of Program.cs:
- using System;
- using Microsoft.AspNetCore.Hosting;
- using Microsoft.AspNetCore.Builder;
- using Microsoft.Extensions.Configuration;
- namespace HelloWorld

  ```
  {
      class Program
      {
      static void Main(string[] args)
      {
           var config = new ConfigurationBuilder()
          .AddCommandLine(args)
          .Build();
          var host = new WebHostBuilder()
          .UseKestrel()
          .UseStartup<Startup>()
         .UseConfiguration(config)
              .Build();
              host.Run();
              }
      }
  }
  ```
- The first thing we do is initialize the configuration sub-system.
- We can use the ConfigurationBuilder to accept configuration settings from JSON files, from environment variables and from the command line.
- We then use the WebHostBuilder class to set up our web host.
- We're using a cross-platform, bootstrapped web server called Kestrel.
-
- 3: Adding a Startup Class and Middleware:
- (same as global.asax.cs file).
- With ASP.NET Core, we can use the UseStartup<> generic method to define a startup class.
- The startup class is expected to be able to support the following methods:
- A constructor that takes an IHostingEnvironment variable
- The Configure method, used to configure the HTTP request pipeline and the application.

- The ConfigureServices method, used to add scoped services to the system to be made available via dependency injection.
- We need to add a Startup class to our project.
- Content of Startup.cs:
- using Microsoft.AspNetCore.Builder;
- using Microsoft.AspNetCore.Hosting;
- using Microsoft.Extensions.Logging;
- using Microsoft.AspNetCore.Http;
- namespace HelloWorld
- {
  ```
      public class Startup
      {
        public Startup(IHostingEnvironment env)
       { }
      public void Configure(IApplicationBuilder app, IHostingEnvironment env,
      ILoggerFactory  loggerFactory)
      {
          app.Run(async (context) =>
          { await context.Response.WriteAsync("Hello, world!"); }
                      );
      }
  ```
- }
- The Use method adds middleware to the HTTP request processing pipeline.
- ASP.NET Core middleware components (request processors) are set up as a chain or pipeline and are given a chance to perform their processing in sequence during each request.
- It is the responsibility of the middleware component to invoke the next component in the sequence or terminate the pipeline if appropriate.
- 
- Middleware components can be added to request processing using the following three methods:
- 1) Map
- Map adds the capability to branch a request pipeline by mapping a specific request path to a handler.
- You can also get even more powerful functionality with the MapWhen method that supports predicate-based branching.
- 2) Use
- Use adds a middleware component to the pipeline. The component’s code must decide whether to terminate or continue the pipeline.
- 3)Run
- The first middleware component added to the pipeline via Run will terminate the pipeline.
- A component added via Use that doesn’t invoke the next component is identical to Run, and will terminate the pipeline.
- 
- 4: Running the App
- type dotnet run from the command line.
- $ dotnet run