EBook Gratis

APRENDIZAJE .net-core

Free unaffiliated eBook created from
Stack Overflow contributors.

#.net-core

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: -net-core

It is an unofficial and free .net-core ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official .net-core.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con .net-core

Observaciones

.NET Core es una plataforma de desarrollo de propósito general mantenida por Microsoft y la comunidad .NET en GitHub.

Las siguientes características definen mejor .NET Core:

- Implementación flexible: se puede incluir en su aplicación o puede instalarse de lado a lado en toda la máquina.
- Multiplataforma: se ejecuta en Windows, macOS y Linux; Puede ser portado a otros sistemas operativos. Los sistemas operativos (OS), las CPU y los escenarios de aplicaciones compatibles crecerán con el tiempo, proporcionados por Microsoft, otras compañías y personas. .NET también se puede utilizar en dispositivos, en la nube y en escenarios integrados / IoT.
- Herramientas de línea de comandos: todos los escenarios de productos se pueden ejercer en la línea de comandos.
- Compatible: .NET Core es compatible con .NET Framework, Xamarin y Mono, a través de la biblioteca estándar de .NET.
- Código abierto: la plataforma .NET Core es de código abierto, con licencias MIT y Apache 2. La documentación está licenciada bajo CC-BY. .NET Core es un proyecto de la Fundación .NET.
- Compatible con Microsoft: .NET Core es compatible con Microsoft, por .NET Core Support

Composición

.NET Core se compone de las siguientes partes:

- **Un tiempo de ejecución .NET** que proporciona un sistema de tipos, carga de ensamblajes, un recolector de basura, interoperabilidad nativa y otros servicios básicos.
- **Un conjunto de bibliotecas de marcos** que proporcionan tipos de datos primitivos, tipos de composición de aplicaciones y utilidades fundamentales.
- **Un conjunto de herramientas SDK y compiladores de lenguaje** que permiten la experiencia del desarrollador base, disponible en .NET Core SDK.
- **El host de la aplicación 'dotnet'** que lanza las aplicaciones .NET Core. El host de la aplicación selecciona y aloja el tiempo de ejecución, proporciona una política de carga de ensamblados e inicia la aplicación. El mismo host también se usa para lanzar herramientas SDK de una manera similar.

(Fuente: documentación oficial).

Versiones

.NET Core

Versión	Fecha de lanzamiento
1.0	2016-06-27
1.1.1	2017-03-07

Examples

Instalación o configuración

Instale .NET Core en macOS 10.11+, después de instalar homebrew:

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

Instale .NET Core SDK desde https://go.microsoft.com/fwlink/?LinkID=835011

Página oficial de Microsoft .NET Core con guías de instalación para Windows, Linux, Mac y Docker

Instrucciones detalladas sobre cómo configurar o instalar .net-core.

Construyendo una aplicación de ejemplo Hello World

Crear un directorio vacío en algún lugar ...

```
mkdir HelloWorld
cd HelloWorld
```

Luego use la tecnología de andamiaje incorporada para crear una muestra de Hello World

```
dotnet new console -o
```

Este comando crea dos archivos:

- `HelloWorld.csproj` describe las dependencias del proyecto, la configuración y el Marco de destino
- `Program.cs` que define el código fuente para el punto de entrada principal y la emisión de la consola de "Hello World".

Si el `dotnet new` comando `dotnet new` falla, asegúrese de haber instalado correctamente .NET Core. Abra el archivo `Program.cs` en su editor favorito para inspeccionarlo:

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

Para restaurar las dependencias del proyecto y el tiempo de ejecución .NET, ejecute

```
dotnet restore
```

Para compilar la aplicación y ejecutarla, ingrese

```
dotnet run
```

Este último comando imprime "Hello World" a la consola.

Instalación desde un archivo binario

Nota: estas instrucciones están dirigidas a .NET Core 1.0.4 & 1.1.1 SDK 1.0.1 y superior.

Cuando utilice archivos binarios para instalar, recomendamos extraer el contenido en / opt / dotnet y crear un enlace simbólico para dotnet. Si ya se instaló una versión anterior de .NET Core, es posible que el directorio y el enlace simbólico ya estén

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

Instalación de Ubuntu

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

Configurar fuente del paquete

El primer paso es establecer la fuente de alimentación para el administrador de paquetes. Esto solo es necesario si no ha configurado previamente la fuente o si está instalando en Ubuntu 16.10 por primera vez.

Ubuntu 14.04 y Linux Mint 17

Comandos

```
sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficmanager.net/repos/dotnet-release/
```

```
trusty main" > /etc/apt/sources.list.d/dotnetdev.list'
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 417A0893
sudo apt-get update
sudo apt-get install dotnet-dev-1.0.1
```

Paquetes instalados

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

Ubuntu 16.04 y Linux Mint 18

Comandos

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

Paquetes instalados

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

Ubuntu 16.10

Comandos

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

Paquetes instalados

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

fuente de documentación oficial

Lea Empezando con .net-core en línea: https://riptutorial.com/es/dot-net-core/topic/1024/empezando-con--net-core

Capítulo 2: .NET Core con Docker

Observaciones

Llene con ejemplos de uso de Docker en la plataforma .NET Core, imágenes base oficiales para la aplicación .NET Core y también la aplicación .NET Core autoubicada

Examples

Dockerfile muestra

La aplicación .NET Core debe publicarse usando la `dotnet publish`

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

Lea .NET Core con Docker en línea: https://riptutorial.com/es/dot-net-core/topic/4224/-net-core-con-docker

Capítulo 3: Comenzando con appsetting.json

Observaciones

Si necesita más información, puede ir y ver la documentación oficial de Microsoft

Examples

Configuración simple

Añade este texto a appsettings.json

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

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

Ahora puedes usar esta configuración en tu aplicación, de la misma manera

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

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

Usando objeto de configuración para ajustes

Crea una clase como una clase debajo

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

```
    public string Key1 { get; set; }
    public int Key2 { get; set; }
}
```

Entonces necesitas agregar este código a tu clase de inicio

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

    public IConfigurationRoot Configuration { get; set; }

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

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

    }
}
```

Entonces podría usarlo en sus controladores de la forma que se presenta a continuación.

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

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

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

Lea Comenzando con appsetting.json en línea: https://riptutorial.com/es/dot-net-core/topic/9057/comenzando-con-appsetting-json

Capítulo 4: Componentes y control de versiones en .NET Core

Introducción

Este documento cubre los diferentes componentes que conforman una distribución de .NET Core y cómo se versionan. Este documento cubre actualmente las versiones 1.x.

Observaciones

Cómo se versionan los componentes en .NET Core.

Componentes

.NET Core consta de varios componentes que se versionan de forma independiente y, a menudo, se pueden combinar y combinar.

- **Marco compartido** . Esto contiene las API y la máquina virtual y otros servicios de tiempo de ejecución necesarios para ejecutar aplicaciones .NET Core.
    - La máquina virtual de .NET actual se llama **CoreCLR** . Esto ejecuta el bytecode .NET al compilarlo JIT y proporciona varios servicios de tiempo de ejecución, incluido un recolector de basura. El código fuente completo de CoreCLR está disponible en https://github.com/dotnet/coreclr .
    - Las API estándar de .NET Core se implementan en **CoreFX** . Esto proporciona implementaciones de todas sus API favoritas, como `System.Runtime` , `System.Theading` y así sucesivamente. El código fuente de CoreFX está disponible en https://github.com/dotnet/corefx .
- **Host** también se llama el *muxer* o *controlador* . Este componente representa el comando `dotnet` y es responsable de decidir qué sucederá a continuación. La fuente para esto está disponible en https://github.com/dotnet/core-setup .
- **SDK** también se llama a veces el *CLI* . Consiste en varias herramientas (subcomandos `dotnet`) y sus implementaciones que tratan con el código de construcción. Esto incluye el manejo de la restauración de dependencias, compilación de código, creación de binarios, producción de paquetes y publicación de paquetes independientes o dependientes del marco. El SDK consta de la *CLI* , que maneja las operaciones de la línea de comandos (en https://github.com/dotnet/cli) y varios subproyectos que implementan las diversas operaciones que la CLI debe realizar.

Componentes en instalaciones .NET Core

Varios paquetes oficiales y no oficiales, tarballs, zip e instaladores para .NET Core (incluidos los disponibles en https://dot.net/core) proporcionan .NET Core en muchas variantes. Dos comunes son SDKs y Runtimes.

Cada *instalación de SDK* o *instalación en tiempo de ejecución* contiene un número (posiblemente 0) de hosts, sdk y componentes de marco compartido descritos anteriormente.

- .NET Core **Runtime** contiene
    - 1 versión de *Shared Framework*
    - 1 versión del *Host*
- .NET Core **SDK** contiene
    - 1 o más versiones del *Marco compartido* (varía según la versión del *SDK*)
    - 1 versión del *Host*
    - 1 versión del *SDK*

Examples

Identificando versiones

Cada componente .NET Core (SDK, Host y Shared Framework) se versiona de forma independiente.

Puedes encontrar la versión para cada uno de ellos por separado.

- **SDK**

  Puede usar la opción `--version` a `dotnet` para ver la versión del SDK. Por ejemplo:

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

  `dotnet --info` también muestra la versión del SDK.

- **Anfitrión**

  Puede ejecutar `dotnet` por sí mismo sin ningún argumento u opción para ver la versión del host.

  ```
  $ ~/dotnet-1.1.1/dotnet

  Microsoft .NET Core Shared Framework Host

    Version  : 1.1.0
  ```

```
  Build    : 362e48a95c86b40cd1f2ef3d08741f7fed897956

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

- **Marco compartido**

  No hay ningún comando actualmente para mostrar los marcos compartidos disponibles. Yo uso `ls /path/to/where/you/installed/dotnet/shared/Microsoft.NETCore.App` que se basa en los detalles de la implementación interna. Por ejemplo:

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

Seleccionando versiones

Es posible tener múltiples .NET Core *SDK* y *Runtimes* disponibles en el disco. Puede seleccionar las versiones para cada uno por separado.

Para seleccionar la versión del SDK a usar, use `global.json` .

Para seleccionar la versión del marco de trabajo compartido que se va a usar, `.csproj` al marco de trabajo especificado en el archivo `.csproj` (o `project.json` si aún lo está usando) .

Lea Componentes y control de versiones en .NET Core en línea: https://riptutorial.com/es/dot-net-core/topic/9592/componentes-y-control-de-versiones-en--net-core

Capítulo 5: Construyendo bibliotecas con .NET Core

Examples

Orientación a .NET estándar

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

Una biblioteca que se dirige a `netstandard1.3` se puede usar en cualquier marco que admita .NET Standard 1.3 **o posterior** . Elegir una versión estándar de .NET inferior para una biblioteca significa que más proyectos pueden usarla, pero hay menos API disponibles.

Orientación tanto a .NET Standard como a .NET Framework 4.x

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

Al dirigirse tanto a `net40` como a `netstandard1.3` , la biblioteca funcionará tanto en proyectos .NET 4.0+ como en proyectos estándar .NET. Es importante mover la dependencia de `NETStandard.Library` a la sección `netstandard1.3` para que solo se haga referencia al construir ese marco.

Producir un paquete NuGet para una biblioteca

Cualquier proyecto que se `netstandard1.X` a `netstandard1.X` se puede empaquetar en un paquete NuGet ejecutando:

```
dotnet pack
```

El paquete resultante se puede cargar en NuGet , MyGet o alojar en una fuente de paquete local .

Dependencias específicas de la plataforma

Puedes especificar diferentes dependencias para cada plataforma:

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

Cuando este proyecto se compila y empaqueta, cada destino del marco utilizará un conjunto diferente de dependencias:

- `net45` (proyectos dirigidos a .NET 4.5+) utilizará el ensamblado `System.Linq` del GAC.
- `netstandard1.3` (proyectos dirigidos .NET Core .NET Standard 1.3) utilizarán la `NETStandard.Library` versión 1.6.0 del paquete NuGet y el `System.Linq` versión preliminar paquete NuGet 4.1.0-RC2.
- `netstandard1.4` usará la misma versión de `NETStandard.Library` , pero la versión 4.1.0 de `System.Linq` .

Agregar una referencia de proyecto a una biblioteca

Si tiene varias bibliotecas en la misma solución, puede agregar referencias locales (proyecto) entre ellas:

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

El valor de la propiedad `target: project` le dice a NuGet que busque en la solución actual para `MyOtherLibrary` , en lugar de en las fuentes de su paquete.

Lea Construyendo bibliotecas con .NET Core en línea: https://riptutorial.com/es/dot-net-core/topic/3290/construyendo-bibliotecas-con--net-core

Capítulo 6: El global.json

Observaciones

El archivo `global.json` es extremadamente poderoso y exclusivo de las aplicaciones `.NET Core` y `ASP.NET Core` .

Examples

Esquema

Tienda de esquemas:

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

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

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

Lea El global.json en línea: https://riptutorial.com/es/dot-net-core/topic/1569/el-global-json

Capítulo 7: Entendiendo System.Runtime vs. mscorlib

Observaciones

Cada biblioteca y lenguaje de programación .NET utiliza un conjunto de tipos de datos elementales como `System.Int32` , `System.Object` , `System.Type` o `System.Uri` . Estos tipos de datos forman la base de todas las demás estructuras, incluidas todas las bibliotecas .NET escritas personalizadas. Todos estos tipos están alojados en una biblioteca base, que es `mscorlib` o `System.Runtime` .

Las bibliotecas que se pueden usar con .NET Core se basan en la biblioteca central de `System.Runtime` , mientras que para .NET Framework (el componente de Windows) se basan en `mscorlib` . Esta diferencia esencial lleva a ...

- la incompatibilidad de las bibliotecas antiguas, ya que esperan un tipo `System.Object, mscorlib` mientras que una biblioteca .NET Core esperaría `System.Object, System.Runtime` .
- una biblioteca de fachada de reenvío de tipos denominada `System.Runtime` to `mscorlib` en **.NET Framework** . De lo contrario, esta biblioteca está (casi) vacía, pero permite el uso de bibliotecas PCL basadas en `System.Runtime` en .NET Framework.
- un tipo que reenvía `mscorlib` al `System.Runtime` en una versión **futura** de **.NET Core** .
- la introducción del concepto de biblioteca de clases portátil (PCL) y, como segunda generación, el `netstandard` como método de unificación entre las dos bibliotecas principales.

Y fuera de eso, innumerables preguntas sobre Stack Overflow.

Examples

Error popular: engañar a NuGet por el camino equivocado

.NET Core `project.json` admite la importación de NuGet (también conocida como esta respuesta SO). Es imposible incluir una biblioteca basada en `mscorlib` debido a una declaración de `import` .

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

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

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

```
    }
}
```

Las importaciones sólo funcionan con bibliotecas portátiles de clase (que se `System.Runtime` basan) o en desuso monikers marco objetivo que también se `System.Runtime` basan (por ejemplo `dotnet` o `dnxcore`)

Error popular: agregue un paquete NuGet que no fue creado para netstandard / netcoreapp (System.Runtime)

En el ejemplo `project.json` continuación, se agregó un conjunto `Microsoft.AspNet.Identity.EntityFramework` que se basa en `mscorlib` .

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

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

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

El autor del ensamblaje `Microsoft.AspNet.Identity.EntityFramework` aún no ha portado el paquete NuGet a `netstandard` (en realidad lo hizo, solo cambió el nombre del paquete como `Microsoft.AspNetCore.Identity.EntityFrameworkCore` ;))

Si encuentra un paquete que necesita y aún no está en la norma de red, comuníquese con el autor y, si es posible, con ayuda para transportarlo.

Error popular: malinterpretando el resultado

Dirigirse a múltiples marcos con `project.json` es simple. Sin embargo el resultado son dos compilaciones diferentes. Tomemos el siguiente ejemplo:

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

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

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

El proceso de compilación del archivo `project.json` conducirá a dos artefactos resultantes:

- Una dll compilada para el mundo `netstandard` basado en el sistema `System.Runtime` que se puede utilizar en .NET Core, .NET Framework (a través de reenviadores de tipo) y productos Xamarin (a través de reenvíos de tipo). Esta dll tiene referencias a `System.Runtime` y `System.Collections.Immutable` .
- Otro dll compilado directamente para .NET Framework basado en `mscorlib` . Esta dll tendrá referencias a `mscorlib` y `System.Collection.Immutable` .

Sin embargo, es importante entender que el `netstandard1.0` basado en `netstandard1.0` `System.Collections.Immutable` utilizará diferentes implementaciones de `System.Runtime` para cada dll de compilación en tiempo de ejecución. El `System.Runtime` que viene con .NET Core no tiene dependencias de ensamblaje por sí mismo (ya que implementa la biblioteca central). El `System.Runtime` utilizado con .NET Framework tiene referencias (para los reenviadores de tipo) a los ensamblados de .NET Framework `mscorlib` , `System.Core` , `System` y `System.ComponentModel.Composition` .

Error popular: agregar accidentalmente una biblioteca mscorlib como dependencias a netstandard / netcoreapp

Otro error popular es la referencia de paquetes que no satisfacen todos los marcos en el ámbito global cuando se apuntan múltiples marcos.

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

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

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

La (meta) biblioteca `NETStandard.Library` funciona bien en este ejemplo, ya que se dirige tanto a `netstandard1.3` como a `net451` . Sin embargo, la biblioteca `Microsoft.AspNet.Identity.EntityFramework` solo se dirige a .NET Framework `net` y `mscorlib` y, por lo tanto, no se puede usar para una salida `netstandard` .

Busque una biblioteca (versión) que cubra ambos marcos o agregue la biblioteca en las dependencias condicionales debajo del marco.

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

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

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

En este caso, la biblioteca solo se puede usar en bloques condicionales #ifdef para la compilación `net451` .

Lea Entendiendo System.Runtime vs. mscorlib en línea: https://riptutorial.com/es/dot-net-core/topic/5994/entendiendo-system-runtime-vs--mscorlib

Capítulo 8: Instalación de .NET Core en Linux

Examples

Instalación genérica para distribuciones de linux.

Si tiene una de las distribuciones de Linux compatibles, puede seguir los pasos en el sitio web .NET Core: https://www.microsoft.com/net

Si tiene una distribución no compatible:

Descargue el **.NET Core SDK** de los enlaces, seleccionando la distribución más cercana a la utilizada.

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

Si tiene soporte para paquetes **deb** , puede instalar paquetes **Ubuntu / Debian** .

Si tiene soporte para paquetes **yum** , puede instalar paquetes de **Fedora** .

Asegúrese de que su sistema tenga al menos:

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

Lea Instalación de .NET Core en Linux en línea: https://riptutorial.com/es/dot-net-core/topic/3354/instalacion-de--net-core-en-linux

Capítulo 9: Interfaz de línea de comandos .NET Core

Examples

Crear .NET Core "Hello World" proyecto de consola

Cree un nuevo **project.json** y ejemplo **Program.cs** :

```
dotnet new
```

Restaurar los paquetes necesarios:

```
dotnet restore
```

Compila y ejecuta el ejemplo:

```
dotnet run
```

Publicar y ejecutar un proyecto .NET Core

Ve al directorio **project.json** y publica:

```
dotnet publish
```

Imprimirá el directorio de salida de la operación, ingresará al directorio y ejecutará el proyecto publicado:

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

La carpeta predeterminada será: `<project root>/bin/<configuration>/<target framework>/publish`

Por ejemplo: `example/bin/Debug/netcoreapp1.0/publish`

Si ha construido el proyecto anteriormente, puede publicar usando:

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

Importante: asegúrese de publicar el proyecto desde el mismo usuario que restauró los paquetes o podría publicarlo sin las bibliotecas requeridas.

Puede especificar la configuración con la opción `-c <Configuration>` . Para publicar en modo Release, use `dotnet publish -c Release` .

Andamios otros tipos de proyectos.

Usando `dotnet new` creará una nueva aplicación de consola. Para andamiar otros tipos de proyectos, use el indicador `-t` o `--type` :

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

Las plantillas disponibles varían según el idioma.

Plantillas C

- `console` (por defecto) - Una aplicación de consola.
- `web` - Una aplicación Core ASP.NET.
- `lib` - Una biblioteca de clases.
- `xunittest` - Un proyecto de prueba xUnit.

F # plantillas

- `console` (por defecto) - Una aplicación de consola.
- `lib` - Una biblioteca de clases.

Proyectos de andamios en otros idiomas.

Por defecto, `dotnet new` crea proyectos en C #. Puede usar la `--lang -l` o `--lang` para `--lang` proyectos en otros idiomas:

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

Actualmente, `dotnet new` admite C # y F #.

Creando un paquete NuGet

Para crear un paquete NuGet desde un proyecto, ejecute este comando desde un directorio que contiene **project.json** :

```
dotnet pack
```

El archivo `.nupkg` resultante será nombrado y versionado de acuerdo con las propiedades en **project.json** . Si hay varios marcos específicos en el archivo de proyecto, el paquete los admitirá a todos.

Ejecutando pruebas automatizadas

Ejecutar la `dotnet test` desde una carpeta que contiene un proyecto de prueba iniciará el corredor de prueba. El corredor de prueba descubrirá y ejecutará las pruebas en el proyecto.

Para ser compatible con la `dotnet test` , el archivo **project.json** debe contener una propiedad `testRunner` y una dependencia de un paquete de corredor de prueba compatible:

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

Lea Interfaz de línea de comandos .NET Core en línea: https://riptutorial.com/es/dot-net-core/topic/4477/interfaz-de-linea-de-comandos--net-core

Creditos

S. No	Capítulos	Contributors
1	Empezando con .net-core	4444, alvaro, Community, Eidolon, gbellmann, hmnzr, jnovo, Max Cheetham, Ming Tong, Thomas
2	.NET Core con Docker	hmnzr
3	Comenzando con appsetting.json	Egorikas
4	Componentes y control de versiones en .NET Core	omajid
5	Construyendo bibliotecas con .NET Core	Chris Forbes, hmnzr, Nate Barbettini
6	El global.json	David Pine
7	Entendiendo System.Runtime vs. mscorlib	David Pine, Thomas
8	Instalación de .NET Core en Linux	Stefano d'Antonio
9	Interfaz de línea de comandos .NET Core	Nate Barbettini, Stefano d'Antonio