eBook Gratuit

APPRENEZ .net-core

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#.net-core

Table des matières

À propos

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.

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

Chapitre 1: Démarrer avec .net-core

Remarques

.NET Core est une plate-forme de développement à usage général gérée par Microsoft et la communauté .NET sur GitHub.

Les caractéristiques suivantes définissent mieux .NET Core:

- Déploiement flexible: peut être inclus dans votre application ou installé côte à côte à l'échelle de l'utilisateur ou de la machine.
- Multiplate-forme: s'exécute sur Windows, macOS et Linux; peut être porté sur d'autres systèmes d'exploitation. Les systèmes d'exploitation, les processeurs et les scénarios d'application pris en charge évolueront avec le temps, fournis par Microsoft, d'autres sociétés et des particuliers. .NET peut également être utilisé dans des scénarios de périphérique, de cloud et intégrés / IoT.
- Outils de ligne de commande: tous les scénarios de produit peuvent être exercés sur la ligne de commande.
- Compatible: .NET Core est compatible avec .NET Framework, Xamarin et Mono, via la bibliothèque standard .NET.
- Open source: la plate-forme .NET Core est open source et utilise les licences MIT et Apache 2. La documentation est sous licence CC-BY. .NET Core est un projet .NET Foundation.
- Pris en charge par Microsoft: .NET Core est pris en charge par Microsoft, par le biais du support .NET Core

Composition

.NET Core est composé des parties suivantes:

- **Un environnement d'exécution .NET** qui fournit un système de type, un chargement d'assemblage, un ramasse-miettes, une interopérabilité native et d'autres services de base.
- **Ensemble de bibliothèques d'** infrastructure fournissant des types de données primitifs, des types de composition d'application et des utilitaires fondamentaux.
- **Un ensemble d'outils SDK et de compilateurs de langage** qui permettent l'expérience du développeur de base, disponible dans le SDK .NET Core.
- **L'hôte de l'application 'dotnet'** qui lance les applications .NET Core. L'hôte de l'application sélectionne et héberge le moteur d'exécution, fournit une stratégie de chargement de l'assembly et lance l'application. Le même hôte est également utilisé pour lancer des outils SDK de la même manière.

(Source: documentation officielle .)

Versions

.NET Core

Version	Date de sortie
1.0	2016-06-27
1.1.1	2017-03-07

Examples

Installation ou configuration

Installez .NET Core sur macOS 10.11+, après l'installation de 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/
```

Installez le .NET Core SDK à partir de https://go.microsoft.com/fwlink/?LinkID=835011.

Page officielle Microsoft .NET Core avec des guides d'installation pour Windows, Linux, Mac et Docker

Instructions détaillées sur la configuration ou l'installation de .net-core.

Création d'un exemple d'application Hello World

Créer un répertoire vide quelque part ...

```
mkdir HelloWorld
cd HelloWorld
```

Ensuite, utilisez la technologie d’échafaudage intégrée pour créer un exemple Hello World

```
dotnet new console -o
```

Cette commande crée deux fichiers:

- `HelloWorld.csproj` décrit les dépendances du projet, les paramètres et le cadre cible
- `Program.cs` qui définit le code source du point d’entrée principal et la console émettant "Hello World".

Si la `dotnet new` commande `dotnet new` échoue, assurez-vous d'avoir correctement installé .NET Core. Ouvrez le fichier `Program.cs` dans votre éditeur préféré pour l'inspecter:

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

Pour restaurer les dépendances du projet et le runtime .NET, exécutez

```
dotnet restore
```

Pour compiler l'application et l'exécuter, entrez

```
dotnet run
```

Cette dernière commande affiche "Hello World" sur la console.

Installation à partir d'une archive binaire

Remarque: ces instructions sont destinées à .NET Core 1.0.4 et 1.1.1 SDK 1.0.1 et versions ultérieures.

Lors de l'utilisation d'archives binaires à installer, il est recommandé d'extraire le contenu dans / opt / dotnet et de créer un lien symbolique pour dotnet. Si une version antérieure de .NET Core est déjà installée, le répertoire et le lien symbolique peuvent déjà

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

Installation d'Ubuntu

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

Configurer la source du paquet

La première étape consiste à établir le flux source pour le gestionnaire de paquets. Ceci n'est nécessaire que si vous n'avez pas encore configuré le source ou si vous effectuez l'installation pour la première fois sur Ubuntu 16.10.

Ubuntu 14.04 et Linux Mint 17

Commandes

```
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
```

Paquets installés

```
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 et Linux Mint 18

Commandes

```
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
```

Paquets installés

```
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

Commandes

```
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
```

Paquets installés

```
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
```

source Documentation officielle

Lire Démarrer avec .net-core en ligne: https://riptutorial.com/fr/dot-net-core/topic/1024/demarrer-avec--net-core

Chapitre 2: .NET Core avec Docker

Remarques

Complétez avec des exemples d'utilisation de Docker sur la plate-forme .NET Core, des images de base officielles pour l'application .NET Core et l'application .NET Core auto-hébergée.

Examples

Dockerfile sample

L'application .NET Core devrait être publiée en utilisant `dotnet publish`

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

Lire .NET Core avec Docker en ligne: https://riptutorial.com/fr/dot-net-core/topic/4224/-net-core-avec-docker

Chapitre 3: Composants et gestion des versions dans .NET Core

Introduction

Ce document couvre les différents composants constituant une distribution .NET Core et leur version. Ce document couvre actuellement les versions 1.x.

Remarques

Comment les composants de .NET Core sont versionnés.

Composants

.NET Core se compose de plusieurs composants dont chaque version est indépendante et peut souvent être mélangée et appariée.

- **Cadre partagé** Cela contient les API, la machine virtuelle et les autres services d'exécution nécessaires à l'exécution des applications .NET Core.
    - La machine virtuelle .NET Core actuelle s'appelle **CoreCLR** . Cela exécute le bytecode .NET en le compilant JIT et fournit divers services d'exécution, y compris un ramasse-miettes. Le code source complet de CoreCLR est disponible sur https://github.com/dotnet/coreclr .
    - Les API standard du noyau .NET sont implémentées dans **CoreFX** . Cela fournit des implémentations de toutes vos API préférées telles que `System.Runtime` , `System.Theading` , etc. Le code source de CoreFX est disponible sur https://github.com/dotnet/corefx .
- **Host** est également appelé le *multiplexeur* ou le *pilote* . Ce composant représente la commande `dotnet` et est chargé de décider de ce qui va se `dotnet` ensuite. La source de cette information est disponible sur https://github.com/dotnet/core-setup .
- **Le SDK** est aussi parfois appelé le *CLI* . Il comprend les différents outils (sous-commandes `dotnet`) et leurs implémentations qui traitent du code du bâtiment. Cela inclut la gestion de la restauration des dépendances, la compilation du code, la création de fichiers binaires, la production de packages et la publication de packages autonomes ou dépendants du framework. Le kit de développement logiciel lui-même consiste en une *interface de* ligne de commande, qui gère les opérations de ligne de commande (à l' adresse https://github.com/dotnet/cli) et divers sous-projets qui implémentent les diverses opérations à effectuer par l'interface de ligne de commande.

Composants dans les installations .NET Core

Divers packages officiels et non officiels, archives tar, zips et programmes d'installation pour .NET Core (y compris ceux disponibles sur https://dot.net/core) fournissent de nombreuses variantes à .NET Core. Les deux principaux sont les SDK et les Runtimes.

Chaque *installation de SDK* ou *installation de Runtime* contient un nombre (éventuellement 0) d'hôtes, de sdk et de composants d'infrastructure partagée décrits ci-dessus.

- .NET Core **Runtime** contient
    - 1 version de *Framework partagé*
    - 1 version de l' *hôte*
- Le **kit de développement** .NET Core contient
    - 1 ou plusieurs versions du *Framework partagé* (varie selon la version du *SDK*)
    - 1 version de l' *hôte*
    - 1 version du *SDK*

Examples

Identifier les versions

Chaque composant .NET Core (SDK, hôte et infrastructure partagée) est versionné indépendamment.

Vous pouvez trouver la version pour chacun d'eux séparément.

- **SDK**

  Vous pouvez utiliser l'option `--version` sur `dotnet` pour voir la version du SDK. Par exemple:

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

  `dotnet --info` montre également la version du SDK.

- **Hôte**

  Vous pouvez exécuter `dotnet` par lui-même sans aucun argument ou option pour voir la version de l'hôte.

  ```
  $ ~/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]
...
```

- **Cadre partagé**

  Il n'y a pas de commande pour afficher les frameworks partagés avaialble. J'utilise `ls /path/to/where/you/installed/dotnet/shared/Microsoft.NETCore.App` qui s'appuie sur les détails de l'implémentation interne. Par exemple:

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

Sélection de versions

Il est possible d'avoir plusieurs *SDK* et *Runtimes* .NET Core disponibles sur disque. Vous pouvez sélectionner les versions pour chacun séparément.

Pour sélectionner la version du SDK à utiliser, utilisez `global.json` .

Pour sélectionner la version de l' `.csproj` partagée à utiliser, ciblez le cadre spécifié dans le fichier `.csproj` (ou `project.json` si vous l'utilisez toujours) .

Lire Composants et gestion des versions dans .NET Core en ligne: https://riptutorial.com/fr/dot-net-core/topic/9592/composants-et-gestion-des-versions-dans--net-core

Chapitre 4: Comprendre System.Runtime vs. mscorlib

Remarques

Chaque bibliothèque et langage de programmation .NET utilise un ensemble de types de données élémentaires tels que `System.Int32` , `System.Object` , `System.Type` ou `System.Uri` . Ces types de données constituent la base de toutes les autres structures, y compris toutes les bibliothèques écrites .NET personnalisées. Tous ces types sont hébergés dans une bibliothèque de base, à savoir `mscorlib` ou `System.Runtime` .

Les bibliothèques pouvant être utilisées avec .NET Core sont basées sur la bibliothèque principale `System.Runtime` , tandis que pour le .NET Framework (composant Windows), elles sont basées sur `mscorlib` . Cette différence essentielle conduit à ...

- l'incompatibilité des anciennes bibliothèques puisqu'elles attendent un type `System.Object, mscorlib` alors qu'une bibliothèque .NET Core attendrait `System.Object, System.Runtime` .
- une bibliothèque de façade de transfert de type appelée `System.Runtime` vers le `mscorlib` du **.NET Framework** . Cette bibliothèque est sinon (presque) vide mais permet l'utilisation de bibliothèques PCL basées sur `System.Runtime` sur le .NET Framework.
- un type de transfert `mscorlib` vers `System.Runtime` dans une **future** version de **.NET Core** .
- l'introduction du concept de bibliothèque de classe portable (PCL) et, en seconde génération, le `netstandard` comme méthode d'unification entre les deux bibliothèques principales.

Et de cela, d'innombrables questions sur le dépassement de pile.

Examples

Erreur populaire: trompeuse NuGet dans le mauvais sens

.NET Core `project.json` prend en charge l'importation de NuGet (alias ment selon cette réponse SO). Il est impossible d'inclure une bibliothèque basée sur `mscorlib` cause d'une instruction d' `import` .

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

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

    "frameworks": {
        "netstandard1.6": {
            "imports": [
```

```
                "net461"
            ]
        }
    }
}
```

Les importations ne fonctionnent qu'avec des bibliothèques de classes portables (basées sur `System.Runtime`) ou des monikers de frameworks cibles obsolètes qui sont également basés sur `System.Runtime` (par exemple, `dotnet` ou `dnxcore`)

Erreur populaire: Ajouter un package NuGet qui n'a pas été créé pour netstandard / netcoreapp (System.Runtime)

Dans l'exemple `project.json` ci-dessous, un assembly `Microsoft.AspNet.Identity.EntityFramework` été ajouté, basé sur `mscorlib` .

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

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

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

L'auteur de l'assembly `Microsoft.AspNet.Identity.EntityFramework` n'a pas encore porté le package NuGet sur `netstandard` (ils l'ont fait, ils ont simplement renommé le package `Microsoft.AspNetCore.Identity.EntityFrameworkCore` ;))

Si vous rencontrez un paquet dont vous avez besoin et que vous n'êtes pas encore sur netstandard, veuillez contacter l'auteur et, si possible, vous aider à le porter.

Erreur populaire: mal comprendre le résultat

Cibler plusieurs frameworks avec `project.json` est simple. Cependant, le résultat est deux compilations différentes. Prenons l'exemple suivant:

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

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

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

```
}
```

Le processus de compilation du fichier `project.json` entraînera deux artefacts résultants:

- Une DLL compilée pour le monde `netstandard` basé sur `System.Runtime` qui peut être utilisée sur .NET Core, .NET Framework (via les redirecteurs de types) et les produits Xamarin (via des redirecteurs de types). Cette DLL contient des références à `System.Runtime` et `System.Collections.Immutable` .
- Une autre DLL compilée directement pour le .NET Framework basé sur `mscorlib` . Cette dll aura des références à `mscorlib` et `System.Collection.Immutable` .

Toutefois, il est important de comprendre que `System.Collections.Immutable` basé sur `netstandard1.0` , utilisera différentes implémentations `System.Runtime` pour chaque DLL de compilation à l'exécution. Le `System.Runtime` fourni avec .NET Core ne possède pas de dépendance d'assemblage (car il implémente la bibliothèque principale). Le `System.Runtime` utilisé avec le .NET Framework contient des références (pour les redirecteurs de types) aux assemblys .NET Framework `mscorlib` , `System.Core` , `System` et `System.ComponentModel.Composition` .

Erreur populaire: Ajout accidentel d'une bibliothèque mscorlib en tant que dépendances à un netstandard / netcoreapp

Une autre erreur populaire est la référence de packages qui ne satisfait pas tous les frameworks sur le périmètre global lorsque plusieurs frameworks sont ciblés.

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

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

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

La bibliothèque (méta) `NETStandard.Library` fonctionne bien dans cet exemple, car elle cible à la fois `netstandard1.3` et `net451` . Toutefois, la bibliothèque `Microsoft.AspNet.Identity.EntityFramework` ne cible que le `net` .NET Framework et `mscorlib` et ne peut donc pas être utilisée pour une sortie `netstandard` .

Recherchez une bibliothèque (version) qui couvre les deux frameworks ou ajoutez la bibliothèque dans les dépendances conditionnelles situées sous le framework.

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

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

```
    },

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

Dans ce cas, la bibliothèque ne peut être utilisée que dans des blocs #ifdef conditionnels pour la construction `net451` .

Lire Comprendre System.Runtime vs. mscorlib en ligne: https://riptutorial.com/fr/dot-net-core/topic/5994/comprendre-system-runtime-vs--mscorlib

Chapitre 5: Création de bibliothèques avec .NET Core

Examples

Ciblant le standard .NET

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

Une bibliothèque qui cible `netstandard1.3` peut être utilisée sur toute `netstandard1.3` charge .NET Standard 1.3 **ou version ultérieure** . Choisir une version inférieure de .NET Standard pour une bibliothèque signifie que davantage de projets peuvent l'utiliser, mais que moins d'API sont disponibles.

Ciblant à la fois .NET Standard et .NET Framework 4.x

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

En ciblant à la fois `net40` et `netstandard1.3` , la bibliothèque fonctionnera dans les projets .NET 4.0+ et les projets .NET Standard. Il est important de déplacer la dépendance `NETStandard.Library` dans la section `netstandard1.3` afin qu'elle soit uniquement référencée lors de la construction de cette structure.

Produire un package NuGet pour une bibliothèque

Tout projet qui cible `netstandard1.X` peut être compressé dans un package NuGet en exécutant:

```
dotnet pack
```

Le package résultant peut être téléchargé sur NuGet , MyGet ou hébergé dans une source de package locale .

Dépendances spécifiques à la plate-forme

Vous pouvez spécifier différentes dépendances pour chaque plate-forme:

```
"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"
    }
}
```

Lorsque ce projet est compilé et compressé, chaque cible de structure utilise un ensemble de dépendances différent:

- `net45` (projets ciblant .NET 4.5+) utilisera l'assembly `System.Linq` du GAC.
- `netstandard1.3` (projets .NET Core ciblant .NET Standard 1.3) utilisera le `NETStandard.Library` version 1.6.0 et le `System.Linq` version préliminaire 4.1.0-rc2.
- `netstandard1.4` utilisera la même version de `NETStandard.Library` , mais la version 4.1.0 de `System.Linq` .

Ajout d'une référence de projet à une bibliothèque

Si vous avez plusieurs bibliothèques dans la même solution, vous pouvez ajouter des références locales (projet) entre elles:

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

La valeur de la propriété `target: project` indique à NuGet de rechercher la solution actuelle de `MyOtherLibrary` , plutôt que dans les sources de votre package.

Lire Création de bibliothèques avec .NET Core en ligne: https://riptutorial.com/fr/dot-net-core/topic/3290/creation-de-bibliotheques-avec--net-core

Chapitre 6: Démarrer avec appsetting.json

Remarques

Si vous avez besoin de plus d'informations, vous pouvez aller voir la documentation officielle de Microsoft

Examples

Configuration simple

Ajouter ce texte à appsettings.json

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

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

Maintenant, vous pouvez utiliser cette configuration dans votre application, comme cela

```
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"]}");
    }
}
```

Utilisation de l'objet de configuration pour les paramètres

Créer une classe comme une classe ci-dessous

```
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; }
}
```

Ensuite, vous devez ajouter ce code à votre classe de démarrage

```
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);

    }
}
```

Ensuite, vous pouvez l'utiliser dans vos contrôleurs de la manière présentée ci-dessous

```
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}");
    }
}
```

Lire Démarrer avec appsetting.json en ligne: https://riptutorial.com/fr/dot-net-core/topic/9057/demarrer-avec-appsetting-json

Chapitre 7: Installation de .NET Core sous Linux

Examples

Installation générique pour les distributions Linux

Si vous avez l'une des distributions Linux prises en charge, vous pouvez suivre les étapes sur le site Web .NET Core: https://www.microsoft.com/net

Si vous avez une distribution non prise en charge:

Téléchargez le **.NET Core SDK à** partir des liens, en sélectionnant la distribution plus proche de celle utilisée.

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

Si vous avez un support pour les paquets **deb** , vous pouvez installer les paquets **Ubuntu / Debian** .

Si vous avez un support pour les paquets **yum** , vous pouvez installer les paquets **Fedora** .

Assurez-vous que votre système a au moins:

```
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
```

Lire Installation de .NET Core sous Linux en ligne: https://riptutorial.com/fr/dot-net-core/topic/3354/installation-de--net-core-sous-linux

Chapitre 8: Interface de ligne de commande .NET Core

Examples

Créer un projet de console .NET Core "Hello World"

Créez un nouveau **project.json** et un exemple **Program.cs** :

```
dotnet new
```

Restaurer les paquets nécessaires:

```
dotnet restore
```

Compilez et exécutez l'exemple:

```
dotnet run
```

Publier et exécuter un projet .NET Core

Accédez au répertoire **project.json** et publiez:

```
dotnet publish
```

Il imprimera le répertoire de sortie de l'opération, entrera dans le répertoire et exécutera le projet publié:

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

Le dossier par défaut sera: `<project root>/bin/<configuration>/<target framework>/publish`

Par exemple: `example/bin/Debug/netcoreapp1.0/publish`

Si vous avez déjà construit le projet, vous pouvez publier en utilisant:

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

Important: Assurez-vous de publier le projet à partir du même utilisateur qui a restauré les packages ou de le publier sans les bibliothèques requises.

Vous pouvez spécifier la configuration avec l'option `-c <Configuration>` . Pour publier en mode Release, utilisez `dotnet publish -c Release` .

Échafaudage d'autres types de projets

L'utilisation de `dotnet new` permettra d'échafauder une nouvelle application console. Pour échafauder d'autres types de projets, utilisez l'indicateur `-t` ou `--type` :

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

Les modèles disponibles varient selon la langue.

Modèles C

- `console` (par défaut) - Une application console.
- `web` - Une application ASP.NET Core.
- `lib` - Une bibliothèque de classes.
- `xunittest` - Un projet de test xUnit.

Modèles F

- `console` (par défaut) - Une application console.
- `lib` - Une bibliothèque de classes.

Projets d'échafaudage dans d'autres langues

Par défaut, `dotnet new` crée des projets C #. Vous pouvez utiliser l'indicateur `-l` ou `--lang` pour échafauder des projets dans d'autres langues:

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

Actuellement, `dotnet new` supporte C # et F #.

Créer un package NuGet

Pour créer un package NuGet à partir d'un projet, exécutez cette commande à partir d'un répertoire contenant **project.json** :

```
dotnet pack
```

Le fichier `.nupkg` résultant sera nommé et versionné selon les propriétés de **project.json** . S'il existe plusieurs frameworks ciblés dans le fichier de projet, le package les supportera tous.

Exécution de tests automatisés

`dotnet test` depuis un dossier contenant un projet de test lancera le testeur. Le testeur découvrira

et exécutera les tests du projet.

Pour être compatible avec le `dotnet test` , le fichier **project.json** doit contenir une propriété `testRunner` et une dépendance sur un package de testeur 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"
}
```

Lire Interface de ligne de commande .NET Core en ligne: https://riptutorial.com/fr/dot-net-core/topic/4477/interface-de-ligne-de-commande--net-core

Chapitre 9: Le global.json

Remarques

Le fichier `global.json` est extrêmement puissant et unique pour les applications `.NET Core` et `ASP.NET Core` .

Examples

Schéma

Magasin de schémas:

```
{
  "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."
        }
      }
    }
  }
}
```

Lire Le global.json en ligne: https://riptutorial.com/fr/dot-net-core/topic/1569/le-global-json

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec .net-core	4444, alvaro, Community, Eidolon, gbellmann, hmnzr, jnovo, Max Cheetham, Ming Tong, Thomas
2	.NET Core avec Docker	hmnzr
3	Composants et gestion des versions dans .NET Core	omajid
4	Comprendre System.Runtime vs. mscorlib	David Pine, Thomas
5	Création de bibliothèques avec .NET Core	Chris Forbes, hmnzr, Nate Barbettini
6	Démarrer avec appsetting.json	Egorikas
7	Installation de .NET Core sous Linux	Stefano d'Antonio
8	Interface de ligne de commande .NET Core	Nate Barbettini, Stefano d'Antonio
9	Le global.json	David Pine