



**.NET Core 3.1**

**Getting Started ガイド**





## 法律上の通知

Copyright © 2020 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux<sup>®</sup> is the registered trademark of Linus Torvalds in the United States and other countries.

Java<sup>®</sup> is a registered trademark of Oracle and/or its affiliates.

XFS<sup>®</sup> is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL<sup>®</sup> is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js<sup>®</sup> is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack<sup>®</sup> Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## 概要

.NET Core は、自動メモリー管理と最新のプログラミング言語を備えた汎用開発プラットフォームです。これにより、ユーザーは高品質のアプリケーションを効率的に構築できます。.NET Core は、認定済みのコンテナを介して Red Hat Enterprise Linux および OpenShift Container Platform で利用できます。.NET Core には次の機能があります。マイクロサービスベースのアプローチに従う機能。一部のコンポーネントは .NET で構築され、他のコンポーネントは Java で構築されますが、すべてが Red Hat Enterprise Linux および OpenShift Container Platform でサポートされている共通プラットフォームで実行できます。Microsoft Windows で新しい .NET Core ワークロードをより簡単に開発する能力。Red Hat Enterprise Linux または Windows Server のいずれかにデプロイして実行できます。異機種環境のデータセンター。基盤となるインフラストラクチャーが

Windows Server にのみ依存することなく .NET アプリケーションを実行できます。 .NET Core 3.1 は、Red Hat Enterprise Linux 7、および OpenShift Container Platform バージョン 3.3 以降でサポートされています。

---

## 目次

<b>第1章 RED HAT ENTERPRISE LINUX での .NET CORE 3.1 の使用</b> .....	<b>3</b>
1.1. RED HAT ENTERPRISE LINUX のインストールおよび登録	3
1.2. .NET CORE のインストール	4
1.3. アプリケーションの作成	5
1.4. アプリケーションの公開	5
1.5. LINUX コンテナでのアプリケーションの実行	6
<b>第2章 RED HAT OPENSIFT CONTAINER PLATFORM での .NET CORE 3.1 の使用</b> .....	<b>8</b>
2.1. イメージストリームのインストール	8
2.2. ソースからアプリケーションのデプロイメント	9
2.3. バイナリーアーティファクトからアプリケーションのデプロイ	10
2.4. JENKINS SLAVE の使用	10
2.5. 環境変数	12
2.6. サンプルアプリケーション	15
<b>第3章 .NET CORE 3.1 への移行</b> .....	<b>17</b>
3.1. .NET CORE の以前のバージョンからの移行	17
3.2. .NET FRAMEWORK から .NET CORE 3.1 への移行	17



# 第1章 RED HAT ENTERPRISE LINUX での .NET CORE 3.1 の使用

本ガイドでは、.NET Core 3.1 を Red Hat Enterprise Linux (RHEL) にインストールする方法を説明します。RHEL 7 の詳細は、[Red Hat Enterprise Linux のドキュメント](#) を参照してください。

## 1.1. RED HAT ENTERPRISE LINUX のインストールおよび登録

1. 以下のイメージのいずれかを使用して RHEL 7 をインストールします。

- [Red Hat Enterprise Linux 7 Server](#)
- [Red Hat Enterprise Linux 7 Workstation](#)
- [Red Hat Enterprise Linux for Scientific Computing](#)  
RHEL のインストール方法は [Red Hat Enterprise Linux インストールガイド](#) を参照してください。

利用可能な RHEL バージョンについては、[Red Hat Enterprise Linux の製品ドキュメントページ](#) を参照してください。

2. 以下のコマンドを使用して、システムを登録します。

```
$ sudo subscription-manager register
```

Red Hat Subscription Management ドキュメントの [Registering and unregistering a System](#) に従って、システムを登録することもできます。

3. システムで利用可能なサブスクリプションの一覧を表示し、サブスクリプションのプール ID を特定します。

```
$ sudo subscription-manager list --available
```

このコマンドは、サブスクリプション名、一意の識別子、有効期限などの詳細を表示します。プール ID は、**Pool ID** で始まる行に一覧表示されます。

4. **dotNET on RHEL** リポジトリへのアクセスを提供するサブスクリプションを割り当てます。直前の手順で特定したプール ID を使用します。

```
$ sudo subscription-manager attach --pool=<appropriate pool ID from the subscription>
```

5. Red Hat Enterprise 7 Server、Red Hat Enterprise 7 Workstation、または HPC Compute Node の .NET Core チャンネルを以下のコマンドのいずれかで有効にします。

```
$ sudo subscription-manager repos --enable=rhel-7-server-dotnet-rpms  
$ sudo subscription-manager repos --enable=rhel-7-workstation-dotnet-rpms  
$ sudo subscription-manager repos --enable=rhel-7-hpc-node-dotnet-rpms
```

6. システムに割り当てられているサブスクリプションの一覧を確認します。

```
$ sudo subscription-manager list --consumed
```

7. **scl** ツールをインストールする

```
$ sudo yum install scl-utils
```



-

## 1.2. .NET CORE のインストール

1. .NET Core 3.1 とそのすべての依存関係をインストールします。

```
$ sudo yum install rh-dotnet31 -y
```

2. bash シェルで **dotnet** コマンドを実行できるように、**rh-dotnet31** Software Collection 環境を有効にします。  
この手順では、最新の 3.1 SDK を使用して .NET Core 3.1 ランタイムをインストールします。新しい SDK が使用可能になると、パッケージの更新として自動的にインストールされます。

```
$ scl enable rh-dotnet31 bash
```

このコマンドは永続化されません。新しいシェルが作成され、**dotnet** コマンドはそのシェル内でのみ利用できます。ログアウト、別のシェルを使用するか、または新しいターミナルを開くと、**dotnet** コマンドが有効にならなくなります。



### 警告

Red Hat は、他のプログラムに影響を及ぼす可能性があるため、**rh-dotnet31** を永続的に有効にすることは推奨していません。たとえば、**rh-dotnet31** には、ベースの RHEL バージョン **libcurl** とは異なるバージョンが含まれます。これにより、**libcurl** の別のバージョンのことが予想されないプログラムで問題が発生する可能性があります。**rh-dotnet** を永続的に有効にする場合は、以下の行を `~/.bashrc` ファイルに追加します。

```
source scl_source enable rh-dotnet31
```

3. 次のコマンドを実行して、インストールの成功を確認します。

```
$ dotnet --info
.NET Core SDK (reflecting any global.json):
  Version: 3.1.100
  Commit: xxxxxxxxxx

Runtime Environment:
  OS Name: rhel
  OS Version: 7
  OS Platform: Linux
  RID: rhel.7-x64
  Base Path: /opt/rh/rh-dotnet31/root/usr/lib64/dotnet/sdk/3.1.100/

Host (useful for support):
  Version: 3.1.0
  Commit: xxxxxxxxxx

.NET Core SDKs installed:
```

```
3.1.100 [/opt/rh/rh-dotnet31/root/usr/lib64/dotnet/sdk]
```

```
.... omitted
```

### 1.3. アプリケーションの作成

1. **hello-world** という名前のディレクトリーに、新しいコンソールアプリケーションを作成します。

```
$ dotnet new console -o hello-world
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on hello-world/hello-world.csproj...
Restore completed in 87.21 ms for /home/<USER>/hello-world/hello-world.csproj.

Restore succeeded.
```

2. プロジェクトを実行します。

```
$ cd hello-world
$ dotnet run
Hello World!
```

### 1.4. アプリケーションの公開

.NET Core 3.1 アプリケーションを公開して、共有されたシステム全体で使用される .NET Core を使用するか、.NET Core を追加できます。この 2 種類のデプロイメントは、それぞれフレームワークに依存するデプロイメント (FDD) および自己完結型デプロイメント (SCD) と呼ばれます。

RHEL では、FDD による公開が推奨されます。この方法により、アプリケーションが Red Hat により構築された最新バージョンの .NET Core を使用し、特定セットのネイティブ依存関係が含まれるようになります。これらのネイティブライブラリーは、**rh-dotnet31** Software Collection に含まれます。一方、SCD は Microsoft が作成したランタイムを使用します。ネイティブライブラリーが使用できなくなるため、**rh-dotnet31** Software Collection 以外でアプリケーションを実行すると問題が発生する可能性があります。

1. フレームワーク依存アプリケーションを公開するには、次のコマンドを使用します。

```
$ dotnet publish -f netcoreapp3.1 -c Release
```

2. 任意: アプリケーションが RHEL 専用の場合は、次のコマンドを使用してその他のプラットフォームに必要な依存関係を削除します。

```
$ dotnet restore -r rhel.7-x64
$ dotnet publish -f netcoreapp3.1 -c Release -r rhel.7-x64 --self-contained false
```

3. Software Collection を有効にし、アプリケーション名を渡して、RHEL システムでアプリケーションを実行します。

```
$ scl enable rh-dotnet31 -- dotnet <app>.dll
```

- このコマンドは、アプリケーションと共に公開されるスクリプトに追加できます。以下のスクリプトをプロジェクトに追加し、**APP** 変数を更新します。

```
#!/bin/bash

APP=<app>
SCL=rh-dotnet31
DIR="$(dirname "$(readlink -f "$0")")"

scl enable $SCL -- "$DIR/$APP" "$@"
```

- パブリッシュ時にスクリプトを含めるには、この **ItemGroup** を **csproj** ファイルに追加します。

```
<ItemGroup>
  <None Update="<scriptname>" Condition=" '$(RuntimeIdentifier)' == 'rhel.7-x64' and
'$(SelfContained)' == 'false'" CopyToPublishDirectory="PreserveNewest" />
</ItemGroup>
```

## 1.5. LINUX コンテナでのアプリケーションの実行

本セクションでは、**dotnet/dotnet-31-runtime-rhel7** イメージを使用して、Linux コンテナ内でプリコンパイルされたアプリケーションを実行する方法を示します。

- mvc\_runtime\_example** という名前のディレクトリーに mvc プロジェクトを新規作成します。

```
$ dotnet new mvc -o mvc_runtime_example
$ cd mvc_runtime_example
```

- プロジェクトを公開します。

```
$ dotnet publish -f netcoreapp3.1 -c Release
```

- Dockerfile** を作成します。

```
$ cat > Dockerfile <<EOF
FROM registry.redhat.io/dotnet/dotnet-31-runtime-rhel7

ADD bin/Release/netcoreapp3.1/publish/ .

CMD ["dotnet", "mvc_runtime_example.dll"]
EOF
```

- イメージを構築します。

```
$ podman build -t dotnet-31-runtime-example .
```



## 注記

**unable to retrieve auth token: invalid username/password** メッセージが含まれるエラーが発生した場合は、**registry.redhat.io** サーバーの認証情報を指定する必要があります。ログインするには、**\$ podman login registry.redhat.io** コマンドを使用します。認証情報は通常、Red Hat カスタマーポータルで使用されるものと同じです。

5. イメージを実行します。

```
$ podman run -d -p8080:8080 dotnet-31-runtime-example
```

6. ブラウザーで欠陥を表示します: <http://127.0.0.1:8080>

## バグの報告

## 第2章 RED HAT OPENSIFT CONTAINER PLATFORM での .NET CORE 3.1 の使用

### 2.1. イメージストリームのインストール

.NET Core イメージストリームは、[s2i-dotnetcore](#) のイメージストリーム定義と OpenShift クライアントバイナリー (**oc**) を使用してインストールされます。イメージストリームの削除、インストール、および更新を容易にするスクリプトが利用できます。

.NET Core イメージストリームは、グローバルな **openshift** 名前空間で定義するか、プロジェクト名前空間でローカルに定義できます。**openshift** 名前空間定義を更新するには、十分なパーミッションが必要です。

RHEL 7 イメージストリームを取得するには、サブスクリプション認証情報を使用して **registry.redhat.io** サーバーに対して認証する必要があります。これらのクレデンシャルは、プルシークレットを OpenShift 名前空間に追加して設定されます。

#### 2.1.1. **oc** を使用したインストール

1. 名前空間にプルシークレットが存在しない場合は、[Red Hat コンテナレジストリーの認証](#)の手順に従ってプルシークレットを追加する必要があります。
2. 以下のコマンドを実行して、利用可能な .NET Core イメージストリームを一覧表示します。

```
$ oc describe is dotnet [-n <namespace>]
```

出力には、インストールされているイメージまたはイメージがインストールされていない場合はメッセージ **Error from server (NotFound)** が表示されます。

3. .NET Core イメージストリームがすでにインストールされている場合は、以下を実行して新しいバージョンを含めることができます。

```
$ oc replace -f https://raw.githubusercontent.com/redhat-developer/s2i-dotnetcore/master/dotnet_imagestreams.json
```

.NET Core のイメージストリームが存在しない場合は、以下を使用してインストールできます。

```
$ oc create -f https://raw.githubusercontent.com/redhat-developer/s2i-dotnetcore/master/dotnet_imagestreams.json
```

#### 2.1.2. スクリプトを使用したインストール

このスクリプトを使用して、.NET Core イメージストリームをインストール、削除、および更新できます。

#### 2.1.3. Linux/macOS

1. [https://raw.githubusercontent.com/redhat-developer/s2i-dotnetcore/master/install\\_imagestreams.sh](https://raw.githubusercontent.com/redhat-developer/s2i-dotnetcore/master/install_imagestreams.sh) からスクリプトをダウンロードします。
2. **oc login** コマンドを使用して、OpenShift クラスターにログインします。

### 3. イメージストリームのインストール/更新

```
./install-imagestreams.sh --os rhel7 [--namespace <namespace>] [--user
<subscription_user> --password <subscription_password>]
```

プルシークレットは、**--user** および **--password** 引数を指定して追加できます。プルシークレットが存在する場合は、指定した引数が無視されます。

このスクリプトの使用に関する詳細は、**./install-imagestreams.sh --help** を実行することができます。

## 2.1.4. Windows

1. <https://raw.githubusercontent.com/redhat-developer/s2i-dotnetcore/master/install-imagestreams.ps1> からスクリプトをダウンロードします。
2. **oc login** コマンドを使用して、OpenShift クラスターにログインします。
3. イメージストリームのインストール/更新

```
./install-imagestreams.sh --OS rhel7 [--Namespace <namespace>] [-User
<subscription_user> -Password <subscription_password>]
```

PowerShell **ExecutionPolicy** はこのスクリプトの実行を禁止する場合があります。ポリシーを緩和するには、**Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass -Force** を実行します。

プルシークレットは **-User** および **-Password** 引数を指定して追加できます。プルシークレットが存在する場合は、指定した引数が無視されます。

このスクリプトの使用に関する詳細は、**Get-Help .\install-imagestreams.ps1** を実行することができます。

## 2.2. ソースからアプリケーションのデプロイメント

1. 以下のコマンドを実行して、**redhat-developer/s2i-dotnetcore-ex** GitHub リポジトリの **dotnetcore-3.1** ブランチの **app** フォルダーにある ASP.NET Core アプリケーションをデプロイします。

```
$ oc new-app --name=exampleapp 'dotnet:3.1~https://github.com/redhat-developer/s2i-dotnetcore-ex#dotnetcore-3.1' --build-env DOTNET_STARTUP_PROJECT=app
```

2. **oc logs** コマンドを使用して、ビルドの進行状況を追跡します。

```
$ oc logs -f bc/exampleapp
```

3. ビルドが完了したら、デプロイされたアプリケーションを表示します。

```
$ oc logs -f dc/exampleapp
```

4. この時点で、プロジェクト内でアプリケーションにアクセスできます。外部からアクセスできるようにするには、**oc expose** コマンドを使用します。その後、**oc get routes** を使用して URL を検索できます。

```
$ oc expose svc/exampleapp
$ oc get routes
```

## 2.3. バイナリーアーティファクトからアプリケーションのデプロイ

.NET Core S2I ビルダーイメージを使用して、提供するバイナリーアーティファクトを使用してアプリケーションをビルドできます。

1. [Publish Applications](#) の説明に従ってアプリケーションを公開します。たとえば、次のコマンドは新しい Web アプリケーションを作成して公開します。

```
$ dotnet new web -o webapp
$ cd webapp
$ dotnet publish -c Release
```

2. **oc new-build** コマンドを使用して、新しいバイナリービルドを作成します。

```
$ oc new-build --name=mywebapp dotnet:3.1 --binary=true
```

3. **oc start-build** コマンドを使用してビルドを開始し、ローカルマシンのバイナリーアーティファクトへのパスを指定します。

```
$ oc start-build mywebapp --from-dir=bin/Release/netcoreapp3.1/publish
```

4. **oc new-app** コマンドを使用して、新しいアプリケーションを作成します。

```
$ oc new-app mywebapp
```

## 2.4. JENKINS SLAVE の使用

OpenShift Container Platform Jenkins イメージは、.NET Core 3.1 スレーブイメージの自動検出を提供します (**dotnet-31**)。自動検出が機能するには、Jenkins スレーブ **ConfigMap** yaml ファイルをプロジェクトに追加する必要があります。

1. Jenkins がデプロイされているプロジェクトに切り替えます。

```
$ oc project <projectname>
```

2. **dotnet-jenkins-slave.yaml** ファイルを作成します。<serviceAccount> 要素に使用される値は、Jenkins スレーブによって使用されるアカウントです。値の指定がない場合は、**default** サービスアカウントが使用されます。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: dotnet-jenkins-slave-31
  labels:
    role: jenkins-slave
data:
  dotnet31: |-
    <org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
    <inheritFrom></inheritFrom>
```

```

<name>dotnet-31</name>
<instanceCap>2247483647</instanceCap>
<idleMinutes>0</idleMinutes>
<label>dotnet-31</label>
<serviceAccount>jenkins</serviceAccount>
<nodeSelector></nodeSelector>
<volumes/>
<containers>
  <org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
    <name>jnlp</name>
    <image>registry.access.redhat.com/dotnet/dotnet-31-jenkins-slave-
rhel7:latest</image>
    <privileged>false</privileged>
    <alwaysPullImage>true</alwaysPullImage>
    <workingDir>/tmp</workingDir>
    <command></command>
    <args>${computer.jnlpMac} ${computer.name}</args>
    <ttyEnabled>false</ttyEnabled>
    <resourceRequestCpu></resourceRequestCpu>
    <resourceRequestMemory></resourceRequestMemory>
    <resourceLimitCpu></resourceLimitCpu>
    <resourceLimitMemory></resourceLimitMemory>
    <envVars/>
  </org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
</containers>
<envVars/>
<annotations/>
<imagePullSecrets/>
<nodeProperties/>
</org.csanchez.jenkins.plugins.kubernetes.PodTemplate>

```

### 3. 設定をプロジェクトにインポートします。

```
$ oc create -f dotnet-jenkins-slave.yaml
```

スレーブイメージが使用されるようになりました。

**例:** 以下の例は、OpenShift Container Platform に Jenkins パイプラインを追加する方法を示しています。Jenkins パイプラインが追加され、Jenkins マスターが実行されていない場合、OpenShift はマスターを自動的にデプロイします。Jenkins サーバーインスタンスのデプロイおよび設定に関する詳細は、[OpenShift Container Platform および Jenkins](#) を参照してください。

手順例の **BuildConfig** yaml ファイルには、**dotnet-31** Jenkins スレーブを使用して設定された単純な Jenkins パイプラインが含まれます。サンプル **BuildConfig** yaml ファイルには 3 つの段階があります。

- まず、ソースはチェックアウトされます。
- 次に、アプリケーションが公開されます。
- 3 つ目は、イメージはバイナリービルドを使用してアセンブルされます。バイナリービルドに関する補足情報は、[バイナリーアーティファクトからのアプリケーションのデプロイ](#) を参照してください。

Jenkins master-slave パイプラインの例を設定するには、以下の手順を行います。



1. **buildconfig.yaml** ファイルを作成します。

```
kind: BuildConfig
apiVersion: v1
metadata:
  name: dotnetapp-build
spec:
  strategy:
    type: JenkinsPipeline
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        node("dotnet-3.1") {
          stage('clone sources') {
            sh "git clone https://github.com/redhat-developer/s2i-dotnetcore-ex --branch
dotnetcore-3.1 ."
          }
          stage('publish') {
            dir('app') {
              sh "dotnet publish -c Release"
            }
          }
          stage('create image') {
            dir('app') {
              sh 'oc new-build --name=dotnetapp dotnet:3.1 --binary=true || true'
              sh 'oc start-build dotnetapp --from-dir=bin/Release/netcoreapp3.1/publish --follow'
            }
          }
        }
      }
```

2. **BuildConfig** ファイルを OpenShift にインポートします。

```
$ oc create -f buildconfig.yaml
```

3. OpenShift コンソールを起動します。 **Builds > Pipelines** の順に移動します。 **dotnetapp-build** パイプラインが利用可能である。
4. **Start Pipeline** をクリックします。 Jenkins イメージを最初にダウンロードする必要があるため、ビルドを開始するまでに時間がかかる場合があります。  
ビルド時に、OpenShift コンソールでさまざまなパイプラインステージが完了したかどうかを確認できます。 **View Log** をクリックし、 Jenkins で完了したパイプラインステージを確認することもできます。
5. Jenkins Pipeline ビルドが完了したら、 **Builds > Images** の順に移動します。 **dotnetapp** イメージがビルドされ、利用可能です。

## 2.5. 環境変数

.NET Core イメージは、.NET Core アプリケーションのビルド動作を制御する多数の環境変数に対応しています。これらの変数は、ビルド構成の一部として設定することも、アプリケーションソースコードリポジトリの **.s2i/environment** ファイルに追加することもできます。

変数名	説明	デフォルト
<code>DOTNET_STARTUP_PROJECT</code>	実行するプロジェクトを選択します。これは、プロジェクトファイル ( <b>csproj</b> 、 <b>fsproj</b> など) またはプロジェクトファイルを1つ含むディレクトリーである必要があります。	.
<code>DOTNET_ASSEMBLY_NAME</code>	実行するアセンブリーを選択します。これには .dll 拡張子を含めないでください。これを、 <b>csproj</b> で指定した出力アセンブリー名 (PropertyGroup/AssemblyName) に設定します。	<b>csproj</b> ファイルの名前
<code>DOTNET_PUBLISH_READRYTORUN</code>	<b>true</b> に設定すると、アプリケーションは事前にコンパイルされます。これにより、アプリケーションの読み込み時に JIT が必要な作業量が削減されるため、起動時間が短縮されます。	<b>false</b>
<code>DOTNET_RESTORE_SOURCES</code>	復元操作中使用される NuGet パッケージソースのスペース区切り一覧を指定します。これにより、 <b>NuGet.config</b> ファイルで指定されたすべてのソースが上書きされます。この変数を <b>DOTNET_RESTORE_CONFIGFILE</b> と組み合わせることはできません。	
<code>DOTNET_RESTORE_CONFIGFILE</code>	復元操作に使用する <b>NuGet.Config</b> ファイルを指定します。この変数を <b>DOTNET_RESTORE_SOURCES</b> と組み合わせることはできません。	
<code>DOTNET_TOOLS</code>	アプリをビルドする前にインストールする .NET ツールの一覧を指定します。@<version> でパッケージ名を保留することにより、特定のバージョンをインストールできます。	
<code>DOTNET_NPM_TOOLS</code>	アプリケーションをビルドする前にインストールする NPM パッケージの一覧を指定します。	

変数名	説明	デフォルト
DOTNET_TEST_PROJECTS	テストするテストプロジェクトの一覧を指定します。これは、複数のプロジェクトファイル、またはプロジェクトファイルを1つ含む複数のディレクトリである必要があります。アイテムごとに <b>dotnet test</b> が呼び出されます。	
DOTNET_CONFIGURATION	Debug モードまたは Release モードでアプリケーションを実行します。この値は、 <b>Release</b> または <b>Debug</b> である必要があります。	<b>Release</b>
DOTNET_VERBOSITY	<b>dotnet build</b> コマンドの詳細度を指定します。設定すると、環境変数はビルドの開始時に出力されます。この変数は、msbuild の詳細度 ( <b>q[uiet]</b> 、 <b>m[inimal]</b> 、 <b>n[ormal]</b> 、 <b>d[etailed]</b> 、および <b>diag[nostic]</b> ) のいずれかに設定できます。	
HTTP_PROXY, HTTPS_PROXY	アプリケーションをビルドおよび実行するときに使用される HTTP/HTTPS プロキシを構成します。	
DOTNET_RM_SRC	<b>true</b> に設定すると、ソースコードはイメージに含まれません。	
DOTNET_SSL_DIRS	信頼する追加の SSL 証明書を含むディレクトリ/ファイルの一覧を指定するために使用されます。証明書は、ビルド中に実行する各プロセスと、ビルド後のイメージで実行するすべてのプロセス (ビルドされたアプリケーションを含む) により信頼されます。項目は、絶対パス (/ で始まる) またはソースリポジトリのパス (証明書など) にすることができます。	
NPM_MIRROR	ビルドプロセス中にカスタム NPM レジストリミラーを使用してパッケージをダウンロードします。	
ASPNETCORE_URLS	この変数は <a href="http://*:8080">http://*:8080</a> に設定され、イメージにより公開されるポートを使用するように ASP.NET Core を設定します。これを変更することは推奨されません。	<a href="http://*:8080">http://*:8080</a>

変数名	説明	デフォルト
<b>DOTNET_RESTORE_DISABLE_PARALLEL</b>	true に設定すると、複数のプロジェクトを並行して復元できなくなります。これにより、ビルドコンテナが低い CPU 制限で実行している場合の復元タイムアウトエラーが減少します。	<b>false</b>
<b>DOTNET_INCREMENTAL</b>	true に設定すると、NuGet パッケージは保持され、インクリメンタルビルドに再利用できます。	<b>false</b>
<b>DOTNET_PACK</b>	true に設定すると、パブリッシュされたアプリケーションが含まれる <b>/opt/app-root/app.tar.gz</b> の <b>tar.gz</b> ファイルを作成します。	

## 2.6. サンプルアプリケーション

.NET Core s2i ビルダーでは、2 つのサンプルアプリケーションを使用できます。

### 2.6.1. s2i-dotnetcore-ex

s2i-dotnetcore-ex は、デフォルトの .NET Core MVC テンプレートアプリケーションです。

このアプリケーションは、.NET Core s2i イメージによってサンプルアプリケーションとして使用され、**Try Example** リンクを使用して OpenShift UI から直接作成できます。

アプリケーションは、OpenShift クライアントバイナリー (**oc**) を使用して作成することもできます。

```
# Add the .NET Core application
$ oc new-app dotnet:3.1~https://github.com/redhat-developer/s2i-dotnetcore-ex#dotnetcore-3.1 --context-dir=app

# Make the .NET Core application accessible externally and show the url
$ oc expose service s2i-dotnetcore-ex
$ oc get route s2i-dotnetcore-ex
```

このアプリケーションの詳細は、<https://github.com/redhat-developer/s2i-dotnetcore-ex/tree/dotnetcore-3.1> を参照してください。

### 2.6.2. s2i-dotnetcore-persistent-ex

s2i-dotnetcore-persistent-ex は、PostgreSQL データベースにデータを格納する単純な CRUD .NET Core Web アプリケーションです。

アプリケーションは、次のように OpenShift クライアント **oc** を使用して作成できます。

```
# Add the database
$ oc new-app postgresql-ephemeral
```

```
# Add the .NET Core application
$ oc new-app dotnet:3.1~https://github.com/redhat-developer/s2i-dotnetcore-persistent-ex#dotnetcore-3.1 --context-dir app

# Add envvars from the the postgresql secret, and database service name envvar.
$ oc set env dc/s2i-dotnetcore-persistent-ex --from=secret/postgresql -e database-service=postgresql

# Make the .NET Core application accessible externally and show the url
$ oc expose service s2i-dotnetcore-persistent-ex
$ oc get route s2i-dotnetcore-persistent-ex
```

このアプリケーションの詳細は、<https://github.com/redhat-developer/s2i-dotnetcore-persistent-ex/tree/dotnetcore-3.1> を参照してください。

[バグの報告](#)

## 第3章 .NET CORE 3.1 への移行

この章では、.NET Core 3.1 の移行情報を提供します。

### 3.1. .NET CORE の以前のバージョンからの移行

次の Microsoft の記事を参照して、以前のバージョンの .NET Core から新しいバージョンの .NET Core に移行してください。

- [.NET Core 2.0 から 2.1 への移行](#)
- [ASP.NET Core 2.2 から 3.0 への移行](#)
- [ASP.NET Core 2.1 から 2.2 への移行](#)
- [への移行](#)

### 3.2. .NET FRAMEWORK から .NET CORE 3.1 への移行

.NET Framework から移行するには、次の情報を確認してください。

#### 3.2.1. 移行に関する考慮事項

.NET Framework に存在するいくつかの技術と API は、.NET Core では使用できません。アプリケーションまたはライブラリーにこれらの API が必要な場合は、代替を見つけることを検討するか、.NET Framework の使用を継続してください。.NET Core は、次の技術と API をサポートしていません。

- Windows Communication Foundation (WCF) サーバー (WCF クライアントがサポートされています)
- .NET リモート処理

さらに、多くの .NET API は、Microsoft Windows 環境でのみ使用できます。次の一覧に、この Windows 固有の API の例をいくつか示します。

- Microsoft.Win32.Registry
- System.AppDomains
- System.Security.Principal.Windows

[.NET Portability Analyzer](#) を使用して、API の相違点と交換の可能性を特定することを検討してください。たとえば、次のコマンドを入力して、.NET Framework 4.6 アプリケーションで使用される API が .NET Core 2.1 でどの程度サポートされているかを確認します。

```
$ dotnet /path/to/ApiPort.dll analyze -f . -r html --target '.NET Framework,Version=4.6' --target '.NET Core,Version=2.1'
```



## 重要

.NET Core の追加設定なしのバージョンでサポートされないいくつかの API は、[Microsoft.Windows.Compatibility](#) nuget パッケージで利用できます。この nuget パッケージを使用するときは注意してください。提供されている API の一部 (Microsoft.Win32.Registry など) は Windows でのみ動作し、アプリケーションが Red Hat Enterprise Linux と互換性がないようにします。

### 3.2.2. .NET Framework の移行に関する記事

.NET Framework から移行する場合は、次の Microsoft の記事を参照してください。

- 一般的なガイドラインは、[「Porting to .NET Core from .NET Framework」](#) を参照してください。
- ライブラリの移植は、[「Porting to .NET Core - Libraries」](#) を参照してください。
- ASP.NET Core への移行は、[「Migrating to ASP.NET Core」](#) を参照してください。

### [バグの報告](#)