Red Hat Enterprise Linux 8

Developing .NET applications in RHEL 8

Installing and running .NET Core 3.1 for developing .NET applications in Red Hat Enterprise Linux 8

Last Updated: 2020-02-09

Red Hat Enterprise Linux 8 Developing .NET applications in RHEL 8

Installing and running .NET Core 3.1 for developing .NET applications in Red Hat Enterprise Linux 8

Legal Notice

Abstract

.NET Core is a general purpose development platform featuring automatic memory management and modern programming languages. It allows users to build high-quality applications efficiently. .NET Core is available in Red Hat Enterprise Linux and OpenShift Container Platform via certified containers. .NET Core offers the following features: The ability to follow a microservices-based approach, where some components are built with .NET and others with Java, but all can run on a common, supported platform in Red Hat Enterprise Linux and OpenShift Container Platform. The capacity to more easily develop new .NET Core workloads on Microsoft Windows. Customers can deploy and run on either Red Hat Enterprise Linux or Windows Server. A heterogeneous data

center, where the underlying infrastructure is capable of running .NET applications without having to rely solely on Windows Server. .NET Core 3.1 is supported on Red Hat Enterprise Linux 7, Red Hat Enterprise Linux 8, and OpenShift Container Platform versions 3.3 and later.

Table of Contents

PREFACE

This guide describes how to install .NET Core 3.1 on Red Hat Enterprise Linux 8 (RHEL). See the Red Hat Enterprise Linux documentation for more information about RHEL 8.

CHAPTER 1. USING .NET CORE 3.1 ON RED HAT ENTERPRISE LINUX

1.1. INSTALLING .NET CORE

This procedure installs the .NET Core 3.1 runtime with the latest 3.1 SDK. When a newer SDK becomes available, it automatically installs as a package update.

Procedure

.NET Core 3.1 is included in the AppStream repositories for RHEL 8. The AppStream repositories are enabled by default on RHEL 8 systems.

1. Install .NET Core 3.1 and all of its dependencies:

```
$ sudo yum install dotnet-sdk-3.1 -y
```

2. Run the following command to verify the installation:

```
$ dotnet --info
```

1.2. CREATING AN APPLICATION

Procedure

1. Create a new Console application in a directory called **hello-world**:

```
$ 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. Run the project:

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

1.3. PUBLISHING APPLICATIONS

.NET Core 3.1 applications can be published to use a shared system-wide version of .NET Core or to include .NET Core. These two deployment types are called framework-dependent deployment (FDD) and self-contained deployment (SCD), respectively.

For Red Hat Enterprise Linux, we recommend publishing by FDD. This method ensures the application is using an up-to-date version of .NET Core, built by Red Hat, that includes a specific set of native dependencies. On the other hand, SCD uses a runtime built by Microsoft.

Procedure

1. Use the following command to publish a framework-dependent application:

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

2. **Optional:** If the application is only for RHEL, trim out the dependencies needed for other platforms with these commands:

```
$ dotnet restore -r rhel.8-x64
$ dotnet publish -f netcoreapp3.1 -c Release -r rhel.8-x64 --self-contained false
```

1.4. RUNNING APPLICATIONS IN CONTAINERS

This section shows how to use the **ubi8/dotnet-31-runtime** image to run a precompiled application inside a container.

Procedure

1. Create a new MVC project in a directory named **mvc_runtime_example**:

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

2. Publish the project:

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

3. Create the **Dockerfile**:

```
$ cat > Dockerfile <<EOF
FROM registry.access.redhat.com/ubi8/dotnet-31-runtime

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

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

4. Build your image:

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

5. Run your image.

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

6. View the result in a browser: **http://127.0.0.1:8080**.

CHAPTER 2. USING .NET CORE 3.1 ON RED HAT OPENSHIFT CONTAINER PLATFORM

2.1. INSTALLING IMAGE STREAMS

.NET Core image streams are installed using image stream definitions from s2i-dotnetcore with the OpenShift client binary (**oc**). A script is available to facilitate removing, installing, and updating the image streams.

.NET Core image streams can be defined in the global **openshift** namespace or locally in a project namespace. Sufficient permissions are required to update the **openshift** namespace definitions.

2.1.1. Installing image streams using oc

Prerequisites

- An existing pull secret present in the namespace. If no pull secret is present in the namespace, you must add one by following the instructions in Red Hat Container Registry Authentication .

Procedure

1. List the available .NET Core image streams:

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

   The output shows installed images or the message **Error from server (NotFound)** if no images are installed.

2. Install the .NET Core image streams:

   ```
   $ oc create -f https://raw.githubusercontent.com/redhat-developer/s2i-dotnetcore/master/dotnet_imagestreams_rhel8.json
   ```

3. Include newer versions of existing .NET Core image streams:

   ```
   $ oc replace -f https://raw.githubusercontent.com/redhat-developer/s2i-dotnetcore/master/dotnet_imagestreams_rhel8.json
   ```

2.1.2. Installing image streams using a script

The script can be used to install, remove, and update .NET Core image streams.

2.1.2.1. Linux/macOS

Procedure

1. Download the script.
2. Log in to the OpenShift cluster using the **oc login** command.
3. Install or update the image streams:

```
./install-imagestreams.sh --os rhel8 [--namespace <namespace>]  [--user
<subscription_user> --password <subscription_password>]
```

The pull secret can be added by providing the **--user** and **--password** arguments. If a pull secret is already present, these arguments are ignored.

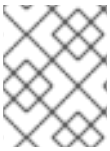
Run **./install-imagestreams.sh --help** for more information on using this script.

2.1.2.2. Windows

Procedure

1. Download the script.
2. Log in to the OpenShift cluster using the **oc login** command.
3. Install or update the image streams:

```
./install-imagestreams.sh --OS rhel8 [--Namespace <namespace>]  [-User
<subscription_user> -Password <subscription_password>]
```



NOTE

The PowerShell **ExecutionPolicy** may prohibit executing this script. To relax the policy, run **Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass -Force**.

The pull secret can be added by providing the **-User** and **-Password** arguments. If a pull secret is already present, these arguments are ignored.

Run **Get-Help .\install-imagestreams.ps1** for more information on using this script.

2.2. DEPLOYING APPLICATIONS

2.2.1. Deploying applications from source

Procedure

1. Run the following commands to deploy the ASP.NET Core application, which is in the **app** folder on the **dotnetcore-3.1** branch of the **redhat-developer/s2i-dotnetcore-ex** GitHub repository:

```
$ 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. Use the **oc logs** command to track progress of the build:

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

3. View the deployed application once the build is finished:

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

4. The application is now accessible within the project. To make the project accessible externally, use the **oc expose** command. You can then use **oc get routes** to find the URL:

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

2.2.2. Deploying applications from binary artifacts

The .NET Core Source-to-Image (S2I) builder image can be used to build an application using binary artifacts that you provide.

Procedure

1. Publish your application as described in Section 1.3, "Publishing applications". For example, the following commands create a new web application and publish it:

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

2. Create a new binary build using the **oc new-build** command:

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

3. Start a build using the **oc start-build** command, specifying the path to the binary artifacts on your local machine:

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

4. Create a new application using the **oc new-app** command:

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

2.3. ENVIRONMENT VARIABLES

The .NET Core images support a number of environment variables to control the build behavior of your .NET Core application. These variables can be set as part of the build configuration, or they can be added to the **.s2i/environment** file in the application source code repository.

Variable Name	Description	Default
DOTNET_STARTUP_PROJECT	Selects the project to run. This must be a project file (for example, **csproj** or **fsproj**) or a folder containing a single project file.	**.**
DOTNET_ASSEMBLY_NAME	Selects the assembly to run. This must not include the **.dll** extension. Set this to the output assembly name specified in **csproj** (PropertyGroup/AssemblyName).	The name of the **csproj** file

Variable Name	Description	Default
DOTNET_PUBLISH_READRYTORUN	When set to **true**, the application will be compiled ahead-of-time. This reduces startup time by reducing the amount of work the JIT needs to do when the application is loading.	**false**
DOTNET_RESTORE_SOURCES	Specifies the space-separated list of NuGet package sources used during the restore operation. This overrides all of the sources specified in the **NuGet.config** file. This variable cannot be combined with **DOTNET_RESTORE_CONFIGFILE**.	
DOTNET_RESTORE_CONFIGFILE	Specifies a **NuGet.Config** file to be used for restore operations. This variable cannot be combined with **DOTNET_RESTORE_SOURCES**.	
DOTNET_TOOLS	Specifies a list of .NET tools to install before building the app. It is possible to install a specific version by post pending the package name with **@<version>**.	
DOTNET_NPM_TOOLS	Specifies a list of NPM packages to install before building the application.	
DOTNET_TEST_PROJECTS	Specifies the list of test projects to test. This must be project files or folders containing a single project file. **dotnet test** is invoked for each item.	
DOTNET_CONFIGURATION	Runs the application in Debug or Release mode. This value should be either **Release** or **Debug**.	**Release**

Variable Name	Description	Default
DOTNET_VERBOSITY	Specifies the verbosity of the **dotnet build** commands. When set, the environment variables are printed at the start of the build. This variable can be set to one of the msbuild verbosity values (**q[uiet]**, **m[inimal]**, **n[ormal]**, **d[etailed]**, and **diag[nostic]**).	
HTTP_PROXY, HTTPS_PROXY	Configures the HTTP or HTTPS proxy used when building and running the application, respectively.	
DOTNET_RM_SRC	When set to **true**, the source code will not be included in the image.	
DOTNET_SSL_DIRS	Used to specify a list of folders or files with additional SSL certificates to trust. The certificates are trusted by each process that runs during the build and all processes that run in the image after the build (including the application that was built). The items can be absolute paths (starting with /) or paths in the source repository (for example, certificates).	
NPM_MIRROR	Uses a custom NPM registry mirror to download packages during the build process.	
ASPNETCORE_URLS	This variable is set to **http://*:8080** to configure ASP.NET Core to use the port exposed by the image. Changing this is not recommended.	**http://*:8080**
DOTNET_RESTORE_DISABLE_PARALLEL	When set to **true**, disables restoring multiple projects in parallel. This reduces restore timeout errors when the build container is running with low CPU limits.	**false**
DOTNET_INCREMENTAL	When set to **true**, the NuGet packages will be kept so they can be re-used for an incremental build.	**false**

Variable Name	Description	Default
DOTNET_PACK	When set to **true**, creates a **tar.gz** file at **/opt/app-root/app.tar.gz** that contains the published application.	

2.4. SAMPLE APPLICATIONS

Two sample applications are available for use with the .NET Core S2I builder.

2.4.1. Creating the MVC sample application

s2i-dotnetcore-ex is the default .NET Core Model, View, Controller (MVC) template application.

This application is used as the example application by the .NET Core S2I image and can be created directly from the OpenShift UI using the *Try Example* link.

The application can also be created with the OpenShift client binary (**oc**).

Procedure

To create the sample application using **oc**:

1. 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
```

2. Make the .NET Core application externally accessible and show the URL:

```
$ oc expose service s2i-dotnetcore-ex
$ oc get route s2i-dotnetcore-ex
```

Additional resources

- For more information about this application, see the **s2i-dotnetcore-ex** repository on GitHub.

2.4.2. Creating the CRUD sample application

s2i-dotnetcore-persistent-ex is a simple Create, Read, Update, Delete (CRUD) .NET Core web application that stores data in a PostgreSQL database.

Procedure

The application can be created using the OpenShift client **oc** as follows:

1. Add the database:

```
$ oc new-app postgresql-ephemeral
```

2. 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
```

3. Add environment variables from the **postgresql** secret and database service name environment variable:

```
$ oc set env dc/s2i-dotnetcore-persistent-ex --from=secret/postgresql -e database-service=postgresql
```

4. Make the .NET Core application externally accessible and show the URL:

```
$ oc expose service s2i-dotnetcore-persistent-ex
$ oc get route s2i-dotnetcore-persistent-ex
```

Additional resources

- For more information about this application, see the **s2i-dotnetcore-persistent-ex** repository on GitHub.

CHAPTER 3. MIGRATION TO .NET CORE 3.1

This chapter provides migration information for .NET Core 3.1.

3.1. MIGRATION FROM PREVIOUS VERSIONS OF .NET CORE

See the following Microsoft articles to migrate from previous versions of .NET Core to newer versions of .NET Core:

- Migrate from .NET Core 2.0 to 2.1
- Migrate from ASP.NET Core 2.2 to 3.0
- Migrate from ASP.NET Core 2.1 to 2.2
- Migrate to ASP.NET Core

3.2. MIGRATION FROM .NET FRAMEWORK TO .NET CORE 3.1

Review the following information to migrate from the .NET Framework.

3.2.1. Migration considerations

Several technologies and APIs present in the .NET Framework are not available in .NET Core. If your application or library requires these APIs, consider finding alternatives or continue using the .NET Framework. .NET Core does not support the following technologies and APIs:

- Windows Communication Foundation (WCF) servers (WCF clients are supported)
- .NET remoting

Additionally, a number of .NET APIs can only be used in Microsoft Windows environments. The following list shows a few examples of these Windows-specific APIs:

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

Consider using the .NET Portability Analyzer to identify API gaps and potential replacements. For example, enter the following command to find out how much of the API used by your .NET Framework 4.6 application is supported by .NET Core 2.1:

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

IMPORTANT

Several APIs that are not supported in the default version of .NET Core may be available from the Microsoft.Windows.Compatibility NuGet package. Be careful when using this NuGet package. Some of the APIs provided (such as **Microsoft.Win32.Registry**) only work on Windows, making your application incompatible with Red Hat Enterprise Linux.

3.2.2. .NET Framework migration articles

Refer to the following Microsoft articles when migrating from .NET Framework.

- For general guidelines, see Porting to .NET Core from .NET Framework .
- For porting libraries, see Porting to .NET Core - Libraries .
- For migrating to ASP.NET Core, see Migrating to ASP.NET Core.