Red Hat support for Spring Boot 2.3

Dekorate Guide for Spring Boot Developers

Use Dekorate to automatically configure your Spring Boot applications for deployment to OpenShift and stand-alone RHEL

Last Updated: 2021-09-09

Red Hat support for Spring Boot 2.3 Dekorate Guide for Spring Boot Developers

Use Dekorate to automatically configure your Spring Boot applications for deployment to OpenShift and stand-alone RHEL

Legal Notice

Abstract

This guide provides details about using Dekorate to automatically generate resource files from your code and prepare your Spring Boot application for deployment to multiple environments.

Table of Contents

PREFACE

Process the code of your Spring Boot application with Dekorate to automatically generate application manifest files and configure your application for deployment to OpenShift.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. To provide feedback, you can highlight the text in a document and add comments.

This section explains how to submit feedback.

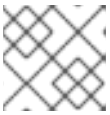
Prerequisites

- You are logged in to the Red Hat Customer Portal.
- In the Red Hat Customer Portal, view the document in **Multi-page HTML** format.

Procedure

To provide your feedback, perform the following steps:

1. Click the **Feedback** button in the top-right corner of the document to see existing feedback.



NOTE

The feedback feature is enabled only in the **Multi-page HTML** format.

2. Highlight the section of the document where you want to provide feedback.
3. Click the **Add Feedback** pop-up that appears near the highlighted text.
   A text box appears in the feedback section on the right side of the page.
4. Enter your feedback in the text box and click **Submit**.
   A documentation issue is created.
5. To view the issue, click the issue tracker link in the feedback view.

CHAPTER 1. CONFIGURING YOUR APPLICATION TO USE SPRING BOOT

Configure your application to use dependencies provided with Red Hat build of Spring Boot. By using the BOM to manage your dependencies, you ensure that your applications always uses the product version of these dependencies that Red Hat provides support for. Reference the Spring Boot BOM (Bill of Materials) artifact in the **pom.xml** file at the root directory of your application. You can use the BOM in your application project in 2 different ways:

- As a dependency in the **<dependencyManagement>** section of the **pom.xml**. When using the BOM as a dependency, your project inherits the version settings for all Spring Boot dependencies from the **<dependencyManagement>** section of the BOM.
- As a parent BOM in the **<parent>** section of the **pom.xml**. When using the BOM as a parent, the **pom.xml** of your project inherits the following configuration values from the parent BOM:
  - versions of all Spring Boot dependencies in the **<dependencyManagement>** section
  - versions plugins in the **<pluginManagement>** section
  - the URLs and names of repositories in the **<repositories>** section
  - the URLs and name of the repository that contains the Spring Boot plugin in the **<pluginRepositories>** section

1.1. PREREQUISITES

- A Maven-based application project that you configure using a **pom.xml** file.
- Access to the Red Hat JBoss Middleware General Availability Maven Repository .

1.2. USING THE SPRING BOOT BOM TO MANAGE DEPENDENCY VERSIONS

Manage versions of Spring Boot product dependencies in your application project using the product BOM.

Procedure

1. Add the **dev.snowdrop:snowdrop-dependencies** artifact to the **<dependencyManagement>** section of the **pom.xml** of your project, and specify the values of the **<type>** and **<scope>** attributes:

```
<project>
 ...
 <dependencyManagement>
  <dependencies>
   <dependency>
    <groupId>dev.snowdrop</groupId>
    <artifactId>snowdrop-dependencies</artifactId>
    <version>2.3.10.Final-redhat-00004</version>
    <type>pom</type>
    <scope>import</scope>
   </dependency>
```

```
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

2. Include the following properties to track the version of the Spring Boot Maven Plugin that you are using:

```
<project>
  ...
  <properties>
    <spring-boot-maven-plugin.version>2.3.10.RELEASE</spring-boot-maven-plugin.version>
  </properties>
  ...
</project>
```

3. Specify the names and URLs of repositories containing the BOM and the supported Spring Boot Starters and the Spring Boot Maven plugin:

```
  <!-- Specify the repositories containing Spring Boot artifacts. -->
  <repositories>
    <repository>
      <id>redhat-ga</id>
      <name>Red Hat GA Repository</name>
      <url>https://maven.repository.redhat.com/ga/</url>
    </repository>
  </repositories>

  <!-- Specify the repositories containing the plugins used to execute the build of your application. -->
  <pluginRepositories>
    <pluginRepository>
      <id>redhat-ga</id>
      <name>Red Hat GA Repository</name>
      <url>https://maven.repository.redhat.com/ga/</url>
    </pluginRepository>
  </pluginRepositories>
```

4. Add **spring-boot-maven-plugin** as the plugin that Maven uses to package your application.

```
<project>
  ...
  <build>
    ...
    <plugins>
      ...
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>${spring-boot-maven-plugin.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
```

```
          </execution>
        </executions>
          <configuration>
            <redeploy>true</redeploy>
          </configuration>
      </plugin>
      ...
    </plugins>
  ...
 </build>
 ...
</project>
```

1.3. USING THE SPRING BOOT BOM TO AS A PARENT BOM OF YOUR APPLICATION

Automatically manage the:

- versions of product dependencies
- version of the Spring Boot Maven plugin
- configuration of Maven repositories containing the product artifacts and plugins

that you use in your application project by including the product Spring Boot BOM as a parent BOM of your project. This method provides an alternative to using the BOM as a dependency of your application.

Procedure

1. Add the **dev.snowdrop:snowdrop-dependencies** artifact to the **<parent>** section of the **pom.xml**:

```
<project>
  ...
  <parent>
    <groupId>dev.snowdrop</groupId>
    <artifactId>snowdrop-dependencies</artifactId>
    <version>2.3.10.Final-redhat-00004</version>
  </parent>
  ...
</project>
```

2. Add **spring-boot-maven-plugin** as the plugin that Maven uses to package your application to the **<build>** section of the **pom.xml**. The plugin version is automatically managed by the parent BOM.

```
<project>
  ...
  <build>
    ...
    <plugins>
      ...
      <plugin>
        <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-maven-plugin</artifactId>
        <executions>
            <execution>
                <goals>
                    <goal>repackage</goal>
                </goals>
            </execution>
        </executions>
        <configuration>
            <redeploy>true</redeploy>
        </configuration>
      </plugin>
      ...
    </plugins>
  ...
 </build>
 ...
</project>
```

1.4. RELATED INFORMATION

- For more information about packaging your Spring Boot application, see the Spring Boot Maven Plugin documentation.

CHAPTER 2. USING DEKORATE IN A SPRING BOOT APPLICATION

Use Dekorate to automatically generate application manifest files and configure your application for deployment to OpenShift.

2.1. OVERVIEW OF DEKORATE

Dekorate is a collection of compile-time annotation processors and application resource generators that are provided with Red Hat build of Spring Boot. It works by parsing annotations in your code when you build your application, and extracting configuration properties. Dekorate then uses the extracted values of properties to generate application configuration resources that you can use to deploy your application to a Kubernetes or OpenShift cluster.

As a developer, you can annotate your code and then use Dekorate to automatically generate application manifests when you build your application, which eliminates the need for you to manually write resource files for deploying your application. When your application is based on a rich application runtime framework, such as Spring Boot, Dekorate can integrate directly with the framework and extract the configuration parameters from the API provided by the framework, thus eliminating the need for you to annotate your code. Dekorate can automatically configure your application by:

- Parsing Dekorate-specific annotations in the application code to obtain value and metadata that are used to populate the manifest files
- Extracting information from configuration resources, such as **application.properties** or **application.yaml**
- Obtaining the necessary metadata from a rich application framework and extracting the configuration values from the **application.properties** or **application.yml** file.

In addition to generating resource definitions for your applications, Dekorate provides hooks allowing you to build and deploy your applications on an OpenShift cluster Dekorate works independently of the language in which you write your applications, and can be used with a wide range of build systems. Dekorate consists of a set of libraries distributed as a Maven BOM. You can add the libraries as dependencies of your application project to use Dekorate with your application.

Red Hat provides support for using Dekorate to generate resource files and hooks that you can use to deploy Java applications based on Spring Boot to OpenShift Container Platform.

2.1.1. Additional resources

- Reference for Dekorate configuration properties for OpenShift .
- Reference for Dekorate configuration properties for Source-to-Image .
- Reference for all Dekorate Configuration properties in the community documentation.

2.2. CONFIGURING YOUR APPLICATION PROJECT TO USE DEKORATE

Add the Dekorate BOM and the OpenShift Annotations Starter to the **pom.xml** file of your application project. Include basic annotations in your source files and package your application with Maven to generate the application manifests.

Prerequisites

- A Maven-based Java application project configured to use Spring Boot
- Java JDK 8 or JDK 11 installed
- Maven installed

Procedure

1. Add the Dekorate OpenShift Spring Starter to the **pom.xml** file of your application to enable Dekorate to porcess your application source code and resource files:

```
<project>
  ...
  <dependencies>
    ...
    <dependency>
      <!-- The OpenShift Spring Starter automatically imports the "io.dekorate:openshift-annotations" dependency. -->
      <groupId>io.dekorate</groupId>
      <artifactId>openshift-spring-starter</artifactId>
    </dependency>
    ...
  </dependencies>
...
<project>
```

2. Add the **@SpringBootApplication** annotation to the main class file of your application project:

```
package org.acme;

@SpringBootApplication
public class Application {
}
```

3. Package your application to process you application code and resource files with Dekorate

```
mvn clean package
```

4. Navigate to the **target/classes/META-INF/dekorate** directory that contains the generated OpenShift manifests.

2.3. CUSTOMIZING YOUR APPLICATION CONFIGURATION WITH DEKORATE

Use Dekorate to customize the configuration of your application for deployment on OpenShift by

- specifying configuration parameters in annotations in the source your application
- setting a property in the **application.properties** file

The following example shows how you can set your application to start with 2 replicas when deployed to OpenShift.

Prerequisites

- A Maven-based Java application project configured to use Spring Boot and Dekorate
- Java JDK 8 or JDK 11 installed
- Maven installed

Procedure

1. Add the Dekorate OpenShift Annotations module as a dependency in the **pom.xml** file of your application:

```
<project>
 ...
 <dependencies>
   ...
  <dependency>
   <groupId>io.dekorate</groupId>
   <artifactId>openshift-spring-starter</artifactId>
  </dependency>
  ...
 </dependencies>
...
<project>
```

2. Configure the default number of replicas that your application starts with when deployed to OpenShift:

   a. Add the **@OpenshiftApplication** annotation to the main source file of your application and set number of replicas to 2. When you build and deploy your application, it automatically starts with 2 replicas of the main application container running:

```
package org.acme;

import io.dekorate.openshift.annotation.OpenshiftApplication;

// include the parameter for the number of replicas to
@OpenshiftApplication(replicas=2)
@SpringBootApplication
public class Application {
}
```

   b. Alternatively, set the **dekorate.openshift.replicas=2** property in the **application.properties** file of your application.

   /src/main/resources/application.properties

```
dekorate.openshift.replicas=2
```

3. Package your application:

```
mvn clean package
```

4. Navigate to the **target/classes/META-INF/dekorate** view the manifests generated by Dekorate. The number of replicas in the deployment configuration YAML template is set to 2:

```
...
spec:
  replicas: 2
  selector:
    matchLabels:
      app: acme
...
```

Additional resources

- Overview of Dekorate configuration properties for OpenShift .

2.4. USING ANNOTATIONLESS CONFIGURATION IN A SPRING BOOT APPLICATION

Use Dekorate to generate OpenShift resource configuration files for your Spring Boot application project by extracting dekorate configuration properties from **application.properties** and **application.yml** files. This method does not require that you annotate your application source, because Dekorate can obtain the required metadata from Spring Boot and the configuration parameters from the property files. Annontationless configuration is a feature of rich framework integration between Spring Boot and Dekorate.

Prerequisites

- A Maven-based application project configured to use Spring Boot and Dekorate
- At least 1 class in your application project annotated with the **@SpringBootApplication** annotation.
- Java JDK 8 or JDK 11 installed
- Maven installed

Procedure

1. Add the following dependencies in the **pom.xml** file of your application:

```
<project>
  ...
  <dependencies>
    ...
    <!-- The OpenShift Spring Starter automatically adds "io.dekorate:openshift-annotations"
as a transitive dependency -->
    <dependency>
      <groupId>io.dekorate</groupId>
      <artifactId>openshfit-spring-starter</artifactId>
    </dependency>
    ...
  </dependencies>
...
<project>
```

2. Add Dekorate configuration properties to the **application.properties** or **application.yml** file in your project. You do not have to add any Dekorate property annotations to your source files.

Note, that you can still use annotations in your source files, but if you do so, Dekorate overwrites parameters provided in annotations with the parameters provided in the **application.properties** or **application.yml** files.

3. Package your application:

```
mvn clean package
```

When you build your application Dekorate parses the configuration in the following resources within your application project. The configuration resources are parsed in an increasing order of priority. This means that if 2 different resources of different type present different values for the same configuration parameter, Dekorate uses the value obtained from a resource that is higher on the list of priorities. For example, if an annotation in your source specifies a parameter value, but a different value is specified for the same parameter in your **application.yml**, Dekorate uses the value it obtains from **application.yml**. Dekorate parses your project resources in the following order of priority:

1. Annotations
2. **application.properties**
3. **application.yaml**
4. **application.yml**

4. Navigate to the **target/classes/META-INF/dekorate** directory that contains the generated **openshift.json** and **openshift.yml** manifest files.

2.5. AUTOMATICALLY EXECUTING OPENSHIFT SOURCE-TO-IMAGE BUILDS WITH DEKORATE

You can use Dekorate to automatically execute an OpenShift container image build after you compile your application with Maven.

Note, that the functionality of automatically triggering Source-to-image builds using Dekorate is available as a Technology Preview. Red Hat does not provide support for using this functionality in a production environment.

Prerequisites

- A Maven-based application project configured to use Spring Boot and Dekorate
- The @SpringBootApplication annotation added to the source files in your project
- Java JDK 8 or JDK 11 installed
- Maven installed
- **oc** command-line tool installed
- You are logged in to an OpenShift cluster using **oc** command-line tool

Procedure

1. Add the Dekorate OpenShift Spring Starter as a dependency to the **pom.xml** file of your application. Note, that this module is included as a transitive dependency in all Dekorate OpenShift Starters:

```
<project>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>io.dekorate</groupId>
      <artifactId>openshift-spring-starter</artifactId>
    </dependency>
    ...
  </dependencies>
...
<project>
```

2. Build and Deploy your application. Include the **-Ddekorate.build=true** property to execute the container image build after Maven compiles your application. Note that the functionality that automatically executes the Source-to-image build is provided as Technology Preview.

```
$ mvn clean install -Ddekorate.build=true
```

You can also execute the Source-to-image build manually from the command line after you compile your application with Maven:

```
# Process your application YAML template that is generated by Dekorate:
$ oc apply -f target/classes/META-INF/dekorate/openshift.yml
# Execute the Source-to-image build and deploy your application to the OpenShift cluster:
$ oc start-build example --from-dir=./target --follow
```

2.6. USING DEKORATE WITH SPRING BOOT ON OPENSHIFT

The following example shows you how:

1. You can use the **openshift-spring-stater** in an application.

2. Dekorate can automaticaly identify the type of the application and configure OpenShift service routes and probes accordingly.

3. You can set up your application to trigger a source-to-image build after Maven compiles your application.

4. Prerequisites

   - A Maven-based application project configured to use Spring Boot and Dekorate
   - The **@SpringBootApplication** annotation added to the source files in your project
   - Java JDK 8 or JDK 11 installed
   - Maven installed
   - **oc** command-line tool installed

- You are logged in to an OpenShift cluster using **oc** command-line tool

Procedure

1. Add the Dekorate Spring Starter as a dependency in the **pom.xml** file of your application project.

   pom.xml

```
<project>
 ...
 <dependencies>
   ...
  <dependency>
   <groupId>io.dekorate</groupId>
   <artifactId>openshift-spring-starter</artifactId>
  </dependency>
  ...
 </dependencies>
...
<project>
```

2. Add the **@SpringBootApplication** annotation to your **Main.java** class. This enables the source-to-image build to start when the application is compiled:

   /src/main/java/io/dekorate/example/sbonopenshift/Main.java

```
package io.dekorate.example.sbonopenshift;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Main {

 public static void main(String[] args) {
  SpringApplication.run(Main.class, args);
 }

}
```

3. Add a Rest controller to your application:

   /src/main/java/io/dekorate/example/sbonopenshift/Controller.java

```
package io.dekorate.example.sbonopenshift;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class Controller {

 @RequestMapping("/")
```

```
  public String hello() {
    return "Hello world";
  }
}
```

The Spring application processor provided by the the Dekorate Spring starter automatically detects the Rest controller and identifies the application type as a web application. For a web application, Dekorate automatically generate the OpenShift application template and configures:

- the OpenShift Service route for your application
- exposes a service on the route of your application
- configures liveness and readiness probe settings

4. Build and deploy your application. Include the **-Ddekorate.deploy=true** property to automatically execute the source-to-image build after Maven compiles your application.

```
mvn clean install -Ddekorate.deploy=true
```

2.7. DEKORATE CONFIGURATION PROPERTIES FOR OPENSHIFT

The properties listed in the table below set the values that Dekorate uses to configure your application for deployment to OpenShift. Dekorate uses the values specified in these properties to populate the Deployment Configuration and application resource files generated for your application project. Each property accepts values of the data type that is listed in the table for the particular property. Some of the properties have a default value that Dekorate uses if you do not specify a value for these attributes. You can set these properties in the **application.properties** file of your application project.

Table 2.1. Dekorate application properties for OpenShift

Property	Data Type	Description	Default Value (if applicable)
dekorate.openshift.part-of	String	The name of the collection of components that your application belongs to. The value of this property is used in the name for other Kubernetes resources that your application contains, such as Deployment Configurations and Services.	If you do not specify a value for this property, Dekorate uses the name of the **groupId** that you use in the Maven project of your application as the default value.

Property	Data Type	Description	Default Value (if applicable)
dekorate.openshift.name	String	The name of the application. The value of this property is used in the name for other Kubernetes resources that your application contains, such as Deployment Configurations and Services.	If you do not specify a value for this property, Dekorate uses the name of the **artifactId** that you use for the Maven project of your application as the default value.
dekorate.openshift.version	String	The version of the application. The value of this property is used in the name of all Kubernetes resources that your application contains, such as Deployment Configurations and Services.	If you do not specify a value for this property, Dekorate uses the **version** that you specify in the Maven project containing your application as the default value.
dekorate.openshift.init-containers	Container[]	Specifies init containers that you want to use in your application	
dekorate.openshift.labels	Label[]	Specifies custom labels to be added to all resources in your application	
dekorate.openshift.annotations	Annotation[]	Specifies custom annotations that you want to add to all resources in your application	
dekorate.openshift.env-vars	Env[]	Specifies environment variables that you want to define for all containers created for your application	
dekorate.openshift.working-dir	String	Specifies the working directory of your application container	

Property	Data Type	Description	Default Value (if applicable)
dekorate.openshift.command	String[]	Specifies commands that you want to use in your container	
dekorate.openshift.arguments	String[]	Specifies custom command-line arguments that you want to use in your container	
dekorate.openshift.replicas	int	Specifies how many replicas of application containers you want to create when you deploy your application	**1**
dekorate.openshift.service-account	String	Specifies the name of the Service account used by your application	
dekorate.openshift.host	String	The name of the host node on which your application is running	
dekorate.openshift.ports	Port[]	Network ports that the services provided by your are exposed on	
dekorate.openshift.service-type	ServiceType	The type of service that is generated for your application	**ClusterIP**
dekorate.openshift.pvc-volumes	PersistentVolumeClaimVolume[]	Persistent Volume Claims that you want to attach to all containers of your application	
dekorate.openshift.secret-volumes	SecretVolume[]	Secret volumes that you want to attach to all containers of your application	
dekorate.openshift.config-map-volumes	ConfigMapVolume[]	ConfigMap volumes that you want to attach to all containers of your application	

Property	Data Type	Description	Default Value (if applicable)
dekorate.openshift.git-repo-volumes	GitRepoVolume[]	Git repository volumes that you want to attach to all containers of your application	
dekorate.openshift.aws-elastic-block-store-volumes	AwsElasticBlockStoreVolume[]	AWS Elastic Block Store volumes that you want to attach to all containers of your application	
dekorate.openshift.azure-disk-volumes	AzureDiskVolume[]	Microsoft Azure disk volumes that you want to attach to all containers of your application	
dekorate.openshift.azure-file-volumes	AzureFileVolume[]	Azure file volumes volumes that you want to attach to all containers of your application	
dekorate.openshift.mounts	Mount[]	Mounts that you want to attach to all containers of your application	
dekorate.openshift.image-pull-policy	ImagePullPolicy	Specify the image pull policy that you want to when deploying your application	**IfNotPresent**
dekorate.openshift.image-pull-secrets	String[]	Specify the image pull secret policy that you want to use when deploying your application	
dekorate.openshift.liveness-probe	Probe	Set up a Liveness probe for your application container	

Property	Data Type	Description	Default Value (if applicable)
dekorate.openshift.readiness-probe	Probe	Set up a Readiness probe for your application container	
dekorate.openshift.request-resources	ResourceRequirements	Specify the amount of resources that your application container requires	
dekorate.openshift.limit-resources	ResourceRequirements	Set a resource limit for your application container	
dekorate.openshift.sidecars	Container[]	Specify containers that you want to deploy as sidecars	
dekorate.openshift.expose	boolean	Set whether you want to expose a Route for your application after you deploy it	**false**
dekorate.openshift.headless	boolean	Set whether you want the service that you generate to execute headless	**false**
dekorate.openshift.auto-deploy-enabled	boolean	Set whether your application is automatically deployed when you generate a deploy hook. Setting this property on your application requires that you hard-code its value in your **application.properties** file. Do not set this property if you want to avoid hard-coding its value. Instead, use the **-Ddekorate.deploy=true** option when deploying your application with Maven	**false**

2.8. DEKORATE CONFIGURATION PROPERTIES FOR SOURCE-TO-IMAGE

The properties listed in the table below set the values that Dekorate uses to configure Source-to-Image (s2i) to build for your applications. You can set these properties in the **application.properties** file of your application project.

Table 2.2. Dekorate configuration properties for S2i

Property	Data Type	Description	Default Value (if applicable)
dekorate.s2i.enabled	boolean	Enable s2i build hook generation for your application	**true**
dekorate.s2i.registry	String	Specify the registry name for the image that you want to build	
dekorate.s2i.group	String	Specify the group ID of the application. This value will be used as the username in the docker image that you build	
dekorate.s2i.name	String	Specify the name of your application. This value is be used as the name of the image that you build.	
dekorate.s2i.version	String	The version of the application. This value is be used as the tag of the image that you build.	
dekorate.s2i.image	String	Specifies the full reference to the image that you want to build. When set, this property overrides the values of the **group**, **name** and **version** properties.	
dekorate.s2i.docker-file	String	Specifies the relative path to the Dockerfile from the root directory of your application project	**Dockerfile**

Property	Data Type	Description	Default Value (if applicable)
dekorate.s2i.builder-image	String	Specifies the name of the S2i builder image that you want to use	**fabric8/s2i-java:2.3**
dekorate.s2i.build-env-vars	Env[]	Set environment variables for the s2i build	
dekorate.s2i.auto-push-enabled	boolean	When **true**, s2i automatically pushes the image to the specified registry when the image is built.	**false**
dekorate.s2i.auto-build-enabled	boolean	When **true**, s2i automatically registers a build hook when the application is compiled	**false**
dekorate.s2i.auto-deploy-enabled	boolean	When **true**, your application is automatically deployed when you generate a deploy hook. Setting this property on your application requires that you hard-code its value in your **application.properties** file. Do not set this property if you want to avoid hard-coding its value. Instead, use the **-Ddekorate.deploy=true** option when deploying your application with Maven	**false**

APPENDIX A. THE SOURCE-TO-IMAGE (S2I) BUILD PROCESS

Source-to-Image (S2I) is a build tool for generating reproducible Docker-formatted container images from online SCM repositories with application sources. With S2I builds, you can easily deliver the latest version of your application into production with shorter build times, decreased resource and network usage, improved security, and a number of other advantages. OpenShift supports multiple build strategies and input sources.

For more information, see the Source-to-Image (S2I) Build chapter of the OpenShift Container Platform documentation.

You must provide three elements to the S2I process to assemble the final container image:

- The application sources hosted in an online SCM repository, such as GitHub.
- The S2I Builder image, which serves as the foundation for the assembled image and provides the ecosystem in which your application is running.
- Optionally, you can also provide environment variables and parameters that are used by S2I scripts.

The process injects your application source and dependencies into the Builder image according to instructions specified in the S2I script, and generates a Docker-formatted container image that runs the assembled application. For more information, check the S2I build requirements, build options and how builds work sections of the OpenShift Container Platform documentation.

APPENDIX B. ADDITIONAL SPRING BOOT RESOURCES

- OpenShift Architecture Overview
- Spring Boot Microservices On Red Hat OpenShift Container Platform 3
- Spring Cloud Kubernetes
- Spring Boot Project
- Spring Framework Project
- OpenShift Spring Boot Lab Microservices
- Creating Spring Boot Applications using Fabric8
- Fabric8 Maven Plugin

APPENDIX C. APPLICATION DEVELOPMENT RESOURCES

For additional information about application development with OpenShift, see:

- OpenShift Interactive Learning Portal

To reduce network load and shorten the build time of your application, set up a Nexus mirror for Maven on your Minishift or CDK:

- Setting Up a Nexus Mirror for Maven

APPENDIX D. PROFICIENCY LEVELS

Each available example teaches concepts that require certain minimum knowledge. This requirement varies by example. The minimum requirements and concepts are organized in several levels of proficiency. In addition to the levels described here, you might need additional information specific to each example.

Foundational

The examples rated at Foundational proficiency generally require no prior knowledge of the subject matter; they provide general awareness and demonstration of key elements, concepts, and terminology. There are no special requirements except those directly mentioned in the description of the example.

Advanced

When using Advanced examples, the assumption is that you are familiar with the common concepts and terminology of the subject area of the example in addition to Kubernetes and OpenShift. You must also be able to perform basic tasks on your own, for example, configuring services and applications, or administering networks. If a service is needed by the example, but configuring it is not in the scope of the example, the assumption is that you have the knowledge to properly configure it, and only the resulting state of the service is described in the documentation.

Expert

Expert examples require the highest level of knowledge of the subject matter. You are expected to perform many tasks based on feature-based documentation and manuals, and the documentation is aimed at most complex scenarios.

APPENDIX E. GLOSSARY

E.1. PRODUCT AND PROJECT NAMES

Developer Launcher (developers.redhat.com/launch)

developers.redhat.com/launch called Developer Launcher is a stand-alone getting started experience provided by Red Hat. It helps you get started with cloud-native development on OpenShift. It contains functional example applications that you can download, build, and deploy on OpenShift.

Minishift or CDK

An OpenShift cluster running on your machine using Minishift.

E.2. TERMS SPECIFIC TO DEVELOPER LAUNCHER

Example

An application specification, for example *a web service with a REST API.*
Examples generally do not specify which language or platform they should run on; the description only contains the intended functionality.

Example application

A language-specific implementation of a particular example on a particular runtime. Example applications are listed in an examples catalog.
For example, an example application is a web service with a REST API implemented using the Thorntail runtime.

Examples Catalog

A Git repository that contains information about example applications.

Runtime

A platform that executes an example application. For example, Thorntail or Eclipse Vert.x.