Red Hat

Red Hat support for Spring Boot 2.4

Spring Boot Runtime Guide

Use Spring Boot 2.4 to develop applications that run on OpenShift and on stand-alone RHEL

Last Updated: 2022-03-11

Red Hat support for Spring Boot 2.4 Spring Boot Runtime Guide

Use Spring Boot 2.4 to develop applications that run on OpenShift and on stand-alone RHEL

Legal Notice

Abstract

This guide provides details about using Spring Boot.

Table of Contents

PREFACE

This guide covers concepts as well as practical details needed by developers to use the Spring Boot runtime. It provides information governing the design of a Spring Boot application deployed as a Linux container on 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. INTRODUCTION TO APPLICATION DEVELOPMENT WITH SPRING BOOT

This section explains the basic concepts of application development with Red Hat runtimes. It also provides an overview about the Spring Boot runtime.

1.1. OVERVIEW OF APPLICATION DEVELOPMENT WITH RED HAT RUNTIMES

Red Hat OpenShift is a container application platform, which provides a collection of cloud-native runtimes. You can use the runtimes to develop, build, and deploy Java or JavaScript applications on OpenShift.

Application development using Red Hat Runtimes for OpenShift includes:

- A collection of runtimes, such as, Eclipse Vert.x, Thorntail, Spring Boot, and so on, designed to run on OpenShift.
- A prescriptive approach to cloud-native development on OpenShift.

OpenShift helps you manage, secure, and automate the deployment and monitoring of your applications. You can break your business problems into smaller microservices and use OpenShift to deploy, monitor, and maintain the microservices. You can implement patterns such as circuit breaker, health check, and service discovery, in your applications.

Cloud-native development takes full advantage of cloud computing.

You can build, deploy, and manage your applications on:

OpenShift Container Platform

A private on-premise cloud by Red Hat.

Red Hat CodeReady Studio

An integrated development environment (IDE) for developing, testing, and deploying applications.

This guide provides detailed information about the Spring Boot runtime. For more information on other runtimes, see the relevant runtime documentation.

1.2. OVERVIEW OF SPRING BOOT

Spring Boot lets you create stand-alone Spring-based applications. See Additional Resources for a list of documents about Spring Boot.

Spring Boot on OpenShift combines streamlined application development capabilities of Spring Boot with the infrastructure and container orchestration functionalities of the OpenShift, such as:

- rolling updates
- service discovery
- canary deployments
- ways to implement common microservice patterns: externalized configuration, health check, circuit breaker, and failover

1.2.1. Spring Boot features and frameworks summary

This guide covers using Spring Boot to develop cloud-native applications on OpenShift. The examples applications in subsequent sections show how to integrate Spring Boot with other Red Hat technologies. You can use these integration capabilities to implement a set of modern design patterns that make your cloud-native Java applications:

- resilient
- responsive
- scalable
- secure

You can choose to build your Spring Boot applications on a regular web server stack or a non-blocking reactive stack.

Red Hat provides support for a release of Spring Boot based on the Snowdrop community project.

The supported runtime framework components include:

- A set of Spring Boot Starters for developing cloud-native Java-based applications on a servlet stack based on Apache Tomcat (Provided with Red Hat Java Web Server product offering) and JBoss Undertow (provided with Red Hat Enterprise Application Platform.)
- A set of Spring Boot Starters for developing cloud-native Java-based applications on a reactive stack using the Spring WebFlux non-blocking API, networking components provided by Eclipse Vert.x, and Reactor Netty.
- Dekorate, a collection of annotation parsers and application template generators for OpenShift and Kubernetes that integrates with Spring Boot. With Dekorate you can automatically create templates that you can use to configure your application for deployment to an OpenShift cluster. When you build your application, Dekorate extracts the configuration parameters from annotations in the source files of your application or from files that contain configuration properties (for example, **application.properties**) in your application project. Dekorate then uses the extracted parameters to create and populate resource files that you can use to deploy your application to an OpenShift cluster. Dekorate works independently of the language and build tools you use, and integrates with multiple cloud-native application frameworks. Red Hat provides support for use of Dekorate to generate application templates for deploying Java-based applications on OpenShift Container Platform. Red Hat provides support for using Dekorate with Maven, other build tools are not supported.

1.2.2. Supported Architectures by Spring Boot

Spring Boot supports the following architectures:

- x86_64 (AMD64)
- IBM Z (s390x) in the OpenShift environment
- IBM Power Systems (ppc64le) in the OpenShift environment

Refer to the section Supported Java images for Spring Boot for more information about the image names.

CHAPTER 2. 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

2.1. 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 .

2.2. 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>2.4.9.Final-redhat-00001</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>2.4.9</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>
```

2.3. 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>2.4.9.Final-redhat-00001</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>
```

2.4. RELATED INFORMATION

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

CHAPTER 3. DEVELOPING AND DEPLOYING A SPRING BOOT RUNTIME APPLICATION

You can create new Spring Boot applications from scratch and deploy them to OpenShift. The recommended approach for specifying and using supported and tested Maven artifacts in a Spring Boot application is to use the OpenShift Application Runtimes Spring Boot BOM.

3.1. DEVELOPING SPRING BOOT APPLICATION

For a basic Spring Boot application, you need to create the following:

- A Java class containing Spring Boot methods.
- A **pom.xml** file containing information required by Maven to build the application.

The following procedure creates a simple **Greeting** application that returns "{"content":"Greetings!"}" as response.

NOTE

For building and deploying your applications to OpenShift, Spring Boot 2.4 only supports builder images based on OpenJDK 8 and OpenJDK 11. Oracle JDK and OpenJDK 9 builder images are not supported.

Prerequisites

- OpenJDK 8 or OpenJDK 11 installed.
- Maven installed.

Procedure

1. Create a new directory **myApp**, and navigate to it.

```
$ mkdir myApp
$ cd myApp
```

   This is the root directory for the application.

2. Create directory structure **src/main/java/com/example/** in the root directory, and navigate to it.

```
$ mkdir -p src/main/java/com/example/
$ cd src/main/java/com/example/
```

3. Create a Java class file **MyApp.java** containing the application code.

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;
```

```
@SpringBootApplication
@RestController
public class MyApp {

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

    @RequestMapping("/")
    @ResponseBody
    public Message displayMessage() {
        return new Message();
    }

    static class Message {
        private String content = "Greetings!";

        public String getContent() {
            return content;
        }

        public void setContent(String content) {
            this.content = content;
        }
    }
}
```

4. Create a **pom.xml** file in the application root directory **myApp** with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <name>MyApp</name>
  <description>My Application</description>

  <!-- Import dependencies from the Spring Boot BOM. -->
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>dev.snowdrop</groupId>
        <artifactId>snowdrop-dependencies</artifactId>
        <version>2.4.9.Final-redhat-00001</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <dependency>
        <groupId>io.dekorate</groupId>
        <artifactId>openshift-spring-starter</artifactId>
      </dependency>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
      </dependency>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
      </dependency>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>2.4.9</version>
      </plugin>
    </plugins>
  </build>

  <!-- 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>

  <pluginRepositories>
    <pluginRepository>
      <id>redhat-ga</id>
      <name>Red Hat GA Repository</name>
      <url>https://maven.repository.redhat.com/ga/</url>
    </pluginRepository>
  </pluginRepositories>

</project>
```

5. Build the application using Maven from the root directory of the application.

```
$ mvn clean package -Popenshift -Ddekorate.deploy=true
```

6. Verify that the application is running.
   Using **curl** or your browser, verify your application is running at **http://localhost:8080**.

```
$ curl http://localhost:8080
{"content":"Greetings!"}
```

3.2. DEPLOYING SPRING BOOT APPLICATION TO OPENSHIFT

To deploy your Spring Boot application to OpenShift, configure the **pom.xml** to use the Dekorate Maven dependency.

You can specify the Dekorate Maven dependency in the **pom.xml** file as follows:

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

You can specify a Java image in the **application.properties** file as follows:

```
dekorate.s2i.builder-image=registry.access.redhat.com/ubi8/openjdk-8:1.3
```

The images are available in the Red Hat Ecosystem Catalog .

3.2.1. Supported Java images for Spring Boot

Spring Boot is certified and tested with various Java images that are available for different operating systems. For example, Java images are available for RHEL 7 and RHEL 8 with OpenJDK 8 or OpenJDK 11.

You require Docker or podman authentication to access the RHEL 8 images in the Red Hat Ecosystem Catalog.

The following table lists the OpenJDK images supported by Spring Boot for different architectures. These container images are available in the Red Hat Ecosystem Catalog . In the catalog, you can search and download the images listed in the table below. The image pages contain authentication procedures required to access the images.

JDK (OS)	Architecture supported	Red Hat Ecosystem Catalog
OpenJDK8 (RHEL 7)	x86_64	redhat-openjdk-18/openjdk18-openshift
OpenJDK11 (RHEL 7)	x86_64	openjdk/openjdk-11-rhel7
OpenJDK8 (RHEL 8)	x86_64	ubi8/openjdk-8-runtime
OpenJDK11 (RHEL 8)	x86_64, IBM Z, and IBM Power Systems	ubi8/openjdk-11

NOTE

The use of a RHEL 8-based container on a RHEL 7 host, for example with OpenShift 3 or OpenShift 4, has limited support. For more information, see the Red Hat Enterprise Linux Container Compatibility Matrix.

3.2.2. Preparing Spring Boot application for OpenShift deployment

In the following procedure, a profile with Dekorate Maven dependency is used for building and deploying the application to OpenShift.

Prerequisites

- Maven is installed.
- Docker or podman authentication into Red Hat Ecosystem Catalog to access RHEL 8 images.

Procedure

1. Add the following content to the **pom.xml** file in the application root directory:

   ```
   ...

   <profiles>
     <profile>
       <id>openshift</id>
       <build>
         <plugins>
           <dependency>
             <groupId>io.dekorate</groupId>
             <artifactId>openshift-spring-starter</artifactId>
           </dependency>
         </plugins>
       </build>
     </profile>
   </profiles>
   ```

2. Set the Java image in the **application.properties** file.
   - x86_64 architecture
     - RHEL 7 with OpenJDK 8

       ```
       dekorate.s2i.builder-image=registry.access.redhat.com/ubi7/openjdk-8:1.3
       ```

     - RHEL 7 with OpenJDK 11

       ```
       dekorate.s2i.builder-image=registry.access.redhat.com/openjdk/openjdk-11-
       rhel7:latest
       ```

     - RHEL 8 with OpenJDK 8

       ```
       dekorate.s2i.builder-image=registry.access.redhat.com/ubi8/openjdk-8:1.3
       ```

     - RHEL 8 with OpenJDK 11

       ```
       dekorate.s2i.builder-image=registry.access.redhat.com/ubi8/openjdk-11:latest
       ```

   - x86_64, IBM Z, and IBM Power System architectures
     - RHEL 8 with OpenJDK 11

```
dekorate.s2i.builder-image=registry.access.redhat.com/ubi8/openjdk-11:latest
```

3.2.3. Deploying Spring Boot application to OpenShift using Dekorate

To deploy your Spring Boot application to OpenShift, you must perform the following:

- Log in to your OpenShift instance.
- Deploy the application to the OpenShift instance.

Prerequisites

- **oc** CLI client installed.
- Maven installed.

Procedure

1. Log in to your OpenShift instance with the **oc** client.

   ```
   $ oc login ...
   ```

2. Create a new project in the OpenShift instance.

   ```
   $ oc new-project MY_PROJECT_NAME
   ```

3. Deploy the application to OpenShift using Maven from the application's root directory. The root directory of an application contains the **pom.xml** file.

   ```
   $ mvn clean package -Popenshift -Ddekorate.deploy=true
   ```

   This command uses Dekorate to launch the S2I process on OpenShift and start the pod.

4. Verify the deployment.

   a. Check the status of your application and ensure your pod is running.

   ```
   $ oc get pods -w
   NAME                       READY    STATUS      RESTARTS   AGE
   MY_APP_NAME-1-aaaaa              1/1      Running     0         58s
   MY_APP_NAME-s2i-1-build         0/1      Completed   0         2m
   ```

   The **MY_APP_NAME-1-aaaaa** pod should have a status of **Running** once it is fully deployed and started.

   Your specific pod name will vary.

   b. Determine the route for the pod.

   Example Route Information

   ```
   $ oc get routes
   NAME              HOST/PORT                                          PATH      SERVICES
   PORT      TERMINATION
   ```

```
MY_APP_NAME       MY_APP_NAME-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME     MY_APP_NAME      8080
```

The route information of a pod gives you the base URL which you use to access it.

In this example, **http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** is the base URL to access the application.

c. Verify that your application is running in OpenShift.

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
{"content":"Greetings!"}
```

3.3. DEPLOYING SPRING BOOT APPLICATION TO STAND-ALONE RED HAT ENTERPRISE LINUX

To deploy your Spring Boot application to stand-alone Red Hat Enterprise Linux, configure the **pom.xml** file in the application, package it using Maven and deploy using the **java -jar** command.

Prerequisites

- RHEL 7 or RHEL 8 installed.

3.3.1. Preparing Spring Boot application for stand-alone Red Hat Enterprise Linux deployment

For deploying your Spring Boot application to stand-alone Red Hat Enterprise Linux, you must first package the application using Maven.

Prerequisites

- Maven installed.

Procedure

1. Add the following content to the **pom.xml** file in the application's root directory:

```
...
<!-- Specify target artifact type for the repackage goal. -->
<packaging>jar</packaging>
...
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>${spring-boot.version}</version>
      <executions>
        <execution>
            <goals>
             <goal>repackage</goal>
           </goals>
        </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
```

2. Package your application using Maven.

   ```
   $ mvn clean package
   ```

   The resulting JAR file is in the **target** directory.

3.3.2. Deploying Spring Boot application to stand-alone Red Hat Enterprise Linux using jar

To deploy your Spring Boot application to stand-alone Red Hat Enterprise Linux, use **java -jar** command.

Prerequisites

- RHEL 7 or RHEL 8 installed.
- OpenJDK 8 or OpenJDK 11 installed.
- A JAR file with the application.

Procedure

1. Deploy the JAR file with the application.

   ```
   $ java -jar my-project-1.0.0.jar
   ```

2. Verify the deployment.
   Use **curl** or your browser to verify your application is running at **http://localhost:8080**:

   ```
   $ curl http://localhost:8080
   ```

CHAPTER 4. DEBUGGING YOUR SPRING BOOT-BASED APPLICATION

This sections contains information about debugging your Spring Boot–based application both in local and remote deployments.

4.1. REMOTE DEBUGGING

To remotely debug an application, you must first configure it to start in a debugging mode, and then attach a debugger to it.

4.1.1. Starting your Spring Boot application locally in debugging mode

One of the ways of debugging a Maven-based project is manually launching the application while specifying a debugging port, and subsequently connecting a remote debugger to that port. This method is applicable at least when launching the application manually using the **mvn spring-boot:run** goal.

Prerequisites

- A Maven-based application

Procedure

1. In a console, navigate to the directory with your application.
2. Launch your application and specify the necessary JVM arguments and the debug port using the following syntax:

   ```
   $ mvn spring-boot:run -Drun.jvmArguments="-Xdebug -
   Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=$PORT_NUMBER"
   ```

   $PORT_NUMBER is an unused port number of your choice. Remember this number for the remote debugger configuration.

   If you want the JVM to pause and wait for remote debugger connection before it starts the application, change **suspend** to **y**.

4.1.2. Starting an uberjar in debugging mode

If you chose to package your application as a Spring Boot uberjar, debug it by executing it with the following parameters.

Prerequisites

- An uberjar with your application

Procedure

1. In a console, navigate to the directory with the uberjar.
2. Execute the uberjar with the following parameters. Ensure that all the parameters are specified before the name of the uberjar on the line.

```
$ java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=$PORT_NUMBER -
jar $UBERJAR_FILENAME
```

$PORT_NUMBER is an unused port number of your choice. Remember this number for the remote debugger configuration.

If you want the JVM to pause and wait for remote debugger connection before it starts the application, change **suspend** to **y**.

4.1.3. Starting your application on OpenShift in debugging mode

To debug your Spring Boot-based application on OpenShift remotely, you must set the **JAVA_DEBUG** environment variable inside the container to **true** and configure port forwarding so that you can connect to your application from a remote debugger.

Prerequisites

- Your application running on OpenShift.
- The **oc** binary installed.
- The ability to execute the **oc port-forward** command in your target OpenShift environment.

Procedure

1. Using the **oc** command, list the available deployment configurations:

   ```
   $ oc get dc
   ```

2. Set the **JAVA_DEBUG** environment variable in the deployment configuration of your application to **true**, which configures the JVM to open the port number **5005** for debugging. For example:

   ```
   $ oc set env dc/MY_APP_NAME JAVA_DEBUG=true
   ```

3. Redeploy the application if it is not set to redeploy automatically on configuration change. For example:

   ```
   $ oc rollout latest dc/MY_APP_NAME
   ```

4. Configure port forwarding from your local machine to the application pod:

   a. List the currently running pods and find one containing your application:

   ```
   $ oc get pod
   NAME                    READY     STATUS      RESTARTS    AGE
   MY_APP_NAME-3-1xrsp     0/1       Running     0           6s
   ...
   ```

   b. Configure port forwarding:

   ```
   $ oc port-forward MY_APP_NAME-3-1xrsp $LOCAL_PORT_NUMBER:5005
   ```

Here, **$LOCAL_PORT_NUMBER** is an unused port number of your choice on your local machine. Remember this number for the remote debugger configuration.

5. When you are done debugging, unset the **JAVA_DEBUG** environment variable in your application pod. For example:

```
$ oc set env dc/MY_APP_NAME JAVA_DEBUG-
```

Additional resources

You can also set the **JAVA_DEBUG_PORT** environment variable if you want to change the debug port from the default, which is **5005**.

4.1.4. Attaching a remote debugger to the application

When your application is configured for debugging, attach a remote debugger of your choice to it. In this guide, Red Hat CodeReady Studio is covered, but the procedure is similar when using other programs.

Prerequisites

- The application running either locally or on OpenShift, and configured for debugging.
- The port number that your application is listening on for debugging.
- Red Hat CodeReady Studio installed on your machine. You can download it from the Red Hat CodeReady Studio download page.

Procedure

1. Start Red Hat CodeReady Studio.
2. Create a new debug configuration for your application:
    a. Click **Run→Debug Configurations**.
    b. In the list of configurations, double-click **Remote Java application.** This creates a new remote debugging configuration.
    c. Enter a suitable name for the configuration in the **Name** field.
    d. Enter the path to the directory with your application into the **Project** field. You can use the **Browse...** button for convenience.
    e. Set the **Connection Type** field to *Standard (Socket Attach)* if it is not already.
    f. Set the **Port** field to the port number that your application is listening on for debugging.
    g. Click **Apply.**
3. Start debugging by clicking the **Debug** button in the Debug Configurations window.
   To quickly launch your debug configuration after the first time, click **Run→Debug History** and select the configuration from the list.

Additional resources

- Debug an OpenShift Java Application with JBoss Developer Studio on Red Hat Knowledgebase.
  Red Hat CodeReady Studio was previously called JBoss Developer Studio.
- A Debugging Java Applications On OpenShift and Kubernetes article on OpenShift Blog.

4.2. DEBUG LOGGING

4.2.1. Add Spring Boot debug logging

Add debug logging to your application.

Prerequisites

- An application that you want to debug.

Procedure

1. Declare a **org.apache.commons.logging.Log** object using the **org.apache.commons.logging.LogFactory** for the class you want to add logging.

   ```
   import org.apache.commons.logging.Log;
   import org.apache.commons.logging.LogFactory;
   ...
   private static Log logger = LogFactory.getLog(TheClass.class);
   ```

2. Add debugging statements using **logger.debug("my logging message")**.

   Example logging statement

   ```
   @GET
   @Path("/greeting")
   @Produces("application/json")
   public Greeting greeting(@QueryParam("name") @DefaultValue("World") String name) {
       String message = String.format(properties.getMessage(), name);

       logger.debug("Message: " + message);

       return new Greeting(message);
   }
   ```

3. Add a **logging.level.fully.qualified.name.of.TheClass=DEBUG** in **src/main/resources/application.properties**.
   For example, if you added a logging statement to **dev.snowdrop.example.service.GreetingEndpoint** you would use:

   ```
   logging.level.dev.snowdrop.example.service.GreetingEndpoint=DEBUG
   ```

   This enables log messages at the **DEBUG** level and above to be shown in the logs for your class.

4.2.2. Accessing Spring Boot debug logs on localhost

Start your application and interact with it to see the debugging statements.

Prerequisites

- An application with debug logging enabled.

Procedure

1. Start your application.

   ```
   $ mvn spring-boot:run
   ```

2. Test your application to invoke debug logging.

   ```
   $ curl http://localhost:8080/api/greeting?name=Sarah
   ```

3. View your application logs to see your debug messages.

   ```
   dev.snowdrop.example.service.GreetingEndpoint     : Message: Hello, Sarah!
   ```

To disable debug logging, remove **logging.level.fully.qualified.name.of.TheClass=DEBUG** from **src/main/resources/application.properties** and restart your application.

4.2.3. Accessing debug logs on OpenShift

Start your application and interact with it to see the debugging statements in OpenShift.

Prerequisites

- The **oc** CLI client installed and authenticated.
- A Maven-based application with debug logging enabled.

Procedure

1. Deploy your application to OpenShift:

   ```
   $ mvn clean package -Popenshift -Ddekorate.deploy=true
   ```

2. View the logs:

   1. Get the name of the pod with your application:

      ```
      $ oc get pods
      ```

   2. Start watching the log output:

      ```
      $ oc logs -f pod/MY_APP_NAME-2-aaaaa
      ```

      Keep the terminal window displaying the log output open so that you can watch the log output.

3. Interact with your application:

   1. Get the route of your application:

```
$ oc get routes
```

2. Make an HTTP request on the **/api/greeting** endpoint of your application:

```
$ curl $APPLICATION_ROUTE/api/greeting?name=Sarah
```

4. Return to the window with your pod logs and inspect debug logging messages in the logs.

```
dev.snowdrop.example.service.GreetingEndpoint     : Message: Hello, Sarah!
```

5. To disable debug logging, remove **logging.level.fully.qualified.name.of.TheClass=DEBUG** from **src/main/resources/application.properties** and redeploy your application.

CHAPTER 5. MONITORING YOUR APPLICATION

This section contains information about monitoring your Spring Boot–based application running on OpenShift.

5.1. ACCESSING JVM METRICS FOR YOUR APPLICATION ON OPENSHIFT

5.1.1. Accessing JVM metrics using Jolokia on OpenShift

Jolokia is a built-in lightweight solution for accessing JMX (Java Management Extension) metrics over HTTP on OpenShift. Jolokia allows you to access CPU, storage, and memory usage data collected by JMX over an HTTP bridge. Jolokia uses a REST interface and JSON-formatted message payloads. It is suitable for monitoring cloud applications thanks to its comparably high speed and low resource requirements.

For Java-based applications, the OpenShift Web console provides the integrated *hawt.io* console that collects and displays all relevant metrics output by the JVM running your application.

Prerequistes

- the **oc** client authenticated
- a Java-based application container running in a project on OpenShift
- latest JDK 1.8.0 image

Procedure

1. List the deployment configurations of the pods inside your project and select the one that corresponds to your application.

```
oc get dc
```

```
NAME        REVISION   DESIRED   CURRENT   TRIGGERED BY
MY_APP_NAME   2          1         1         config,image(my-app:6)
...
```

2. Open the YAML deployment template of the pod running your application for editing.

```
oc edit dc/MY_APP_NAME
```

3. Add the following entry to the **ports** section of the template and save your changes:

```
...
spec:
  ...
  ports:
  - containerPort: 8778
    name: jolokia
    protocol: TCP
  ...
...
```

4. Redeploy the pod running your application.

   ```
   oc rollout latest dc/MY_APP_NAME
   ```

   The pod is redeployed with the updated deployment configuration and exposes the port **8778**.

5. Log into the OpenShift Web console.

6. In the sidebar, navigate to *Applications* > *Pods*, and click on the name of the pod running your application.

7. In the pod details screen, click *Open Java Console* to access the hawt.io console.

Additional resources

- hawt.io documentation

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. UPDATING THE DEPLOYMENT CONFIGURATION OF AN EXAMPLE APPLICATION

The deployment configuration for an example application contains information related to deploying and running the application in OpenShift, such as route information or readiness probe location. The deployment configuration of an example application is stored in a set of YAML files. For the example applications that use Nodeshift, the YAML files are located in the **.nodeshift** directory.

IMPORTANT

The deployment configuration files used by Nodeshift do not have to be full OpenShift resource definitions. Nodeshift can take the deployment configuration files and add some missing information to create a full OpenShift resource definition. The resource definitions generated by Dekorate are available in the **target/classes/META-INF/dekorate/** directory. The resource definitions generated by Nodeshift are available in the **tmp/nodeshift/resource/** directory.

Prerequisites

- An existing example project.
- The **oc** CLI client installed.

Procedure

1. Edit an existing YAML file or create an additional YAML file with your configuration update.
   - For example, if your example already has a YAML file with a **readinessProbe** configured, you could change the **path** value to a different available path to check for readiness:

     ```
     spec:
       template:
         spec:
           containers:
             readinessProbe:
               httpGet:
                 path: /path/to/probe
                 port: 8080
                 scheme: HTTP
     ...
     ```

   - If a **readinessProbe** is not configured in an existing YAML file, you can also create a new YAML file in the same directory with the **readinessProbe** configuration.
2. Deploy the updated version of your example using Maven or npm.
3. Verify that your configuration updates show in the deployed version of your example.

   ```
   $ oc export all --as-template='my-template'

   apiVersion: template.openshift.io/v1
   kind: Template
   metadata:
     creationTimestamp: null
   ```

```
  name: my-template
objects:
- apiVersion: template.openshift.io/v1
  kind: DeploymentConfig
  ...
  spec:
    ...
    template:
      ...
      spec:
        containers:
          ...
          livenessProbe:
            failureThreshold: 3
            httpGet:
              path: /path/to/different/probe
              port: 8080
              scheme: HTTP
            initialDelaySeconds: 60
            periodSeconds: 30
            successThreshold: 1
            timeoutSeconds: 1
          ...
```

Additional resources

If you updated the configuration of your application directly using the web-based console or the **oc** CLI client, export and add these changes to your YAML file. Use the **oc export all** command to show the configuration of your deployed application.

APPENDIX C. DEPLOYING A SPRING BOOT APPLICATION USING WAR FILES

As an alternative to the supported application packaging and deployment workflow using fat JAR files, you can package and deploy a Spring Boot application as a WAR (Web Application Archive) file. You must configure your build and deployment settings to ensure that your application builds and deploys correctly on OpenShift.

Prerequisites

- A Spring Boot application.
- Fabric8 Maven Plugin used to deploy your application to OpenShift.
- Spring Boot Maven Plugin used to package your application.

Procedure

1. Add **war** packaging to the **pom.xml** file of your project:

   Example pom.xml

   ```
   <project ...>
     ...
     <packaging>war</packaging>
     ...
   </project>
   ```

2. Specify **spring-boot-starter-tomcat** as a dependency of your application:

   Example pom.xml

   ```
   <project ...>
     ...
     <dependencies>
       ...
       <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-tomcat</artifactId>
       </dependency>
       ...
     </dependencies>
     ...
   </project>
   ```

3. Ensure the **repackage** Maven goal for the Spring Boot Maven plugin is defined in the **pom.xml** file:

   Example pom.xml

   ```
   <project ...>
   ...
     <build>
   ```

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

This ensures that the Spring Boot classes used to launch the application are included in the WAR file, and that the corresponding properties for these classes are defined in the **MANIFEST.mf** file of the WAR file:

- **Main-Class: org.springframework.boot.loader.WarLauncher**
- **Spring-Boot-Classes: WEB-INF/classes/**
- **Spring-Boot-Lib: WEB-INF/lib/**
- **Spring-Boot-Version: 2.4.9**

4. Add the **ARTIFACT_COPY_ARGS** environment variable to the **pom.xml** file.
The Fabric8 Maven Plugin consumes this variable during the build process and ensures that the *Build and Deploy* tool uses the WAR file (rather than the default fat JAR file) to create the application container image:

Example pom.xml

```
    ...
    <profile>
      <id>openshift</id>
      <build>
        <plugins>
          <plugin>
            <groupId>io.fabric8</groupId>
            <artifactId>fabric8-maven-plugin</artifactId>
            <executions>
              ...
            </executions>
            <configuration>
                <images>
                    <image>
                        <name>${project.artifactId}:%t</name>
                        <alias>${project.artifactId}</alias>
                        <build>
                            <from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-
```

```
openshift:${openjdk18-openshift.version}</from>
                    <assembly>
                      <basedir>/deployments</basedir>
                      <descriptorRef>artifact</descriptorRef>
                    </assembly>
                    <env>
                      <ARTIFACT_COPY_ARGS>*.war</ARTIFACT_COPY_ARGS>
                      <JAVA_APP_DIR>/deployments</JAVA_APP_DIR>
                    </env>
                    <ports>
                      <port>8080</port>
                    </ports>
                  </build>
                </image>
              </images>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
    ...
```

5. Add the **JAVA_APP_JAR** environment variable to the **src/main/fabric8/deployment.yml** file. This variable instructs the Fabric8 Maven Plugin to launch your application using the WAR file included with the container. If **src/main/fabric8/deployment.yml** does not exist, you can create it.

   Example deployment.yml

   ```
   spec:
     template:
       spec:
         containers:
           ...
             env:
             - name: JAVA_APP_JAR
               value: ${project.artifactId}-${project.version}.war
   ```

6. Build and deploy your application:

   ```
   mvn clean fabric8:deploy -Popenshift
   ```

APPENDIX D. 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

APPENDIX E. 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 OpenShift Container Platform:

- Setting Up a Nexus Mirror for Maven

APPENDIX F. 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.