Red Hat JBoss Middleware for OpenShift 3

Red Hat Java S2I for OpenShift

Using Red Hat Java S2I for OpenShift

Last Updated: 2017-11-21

Red Hat JBoss Middleware for OpenShift 3 Red Hat Java S2I for OpenShift

Using Red Hat Java S2I for OpenShift

Legal Notice

Copyright © 2017 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

Guide to using the Red Hat Java S2I for OpenShift

Table of Contents

CHAPTER 1. INTRODUCTION 3
1.1. WHAT IS RED HAT JAVA S2I FOR OPENSHIFT 3

CHAPTER 2. BEFORE YOU BEGIN 4
2.1. INITIAL SETUP 4
2.2. VERSION COMPATIBILITY AND SUPPORT 4

CHAPTER 3. GET STARTED 5
3.1. SOURCE TO IMAGE (S2I) BUILD 5
3.2. BINARY BUILDS 5
3.3. BUILD USING THE WEB CONSOLE 9

CHAPTER 4. TUTORIALS 12
4.1. EXAMPLE WORKFLOW: USING MAVEN TO BUILD AND RUN UBER JAR ON JAVA S2I FOR OPENSHIFT IMAGE 12
4.1.1. Prepare for Deployment 12
4.1.2. Deployment 13
4.1.3. Post-Deployment 13
4.1.3.1. Creating a Route 13
4.2. EXAMPLE WORKFLOW: REMOTE DEBUGGING A JAVA APPLICATION RUNNING ON JAVA S2I FOR OPENSHIFT IMAGE 13
4.2.1. Prepare for Deployment 14
4.2.2. Deployment 14
4.2.2.1. Enabling Remote Debugging for a New Application 14
4.2.2.2. Enabling Remote Debugging for an Existing Application 14
4.2.2.3. Connect Local Debugging Port to a Port on the Pod 14
4.2.3. Post-Deployment 15
4.3. EXAMPLE WORKFLOW: RUNNING FLAT CLASSPATH JAR ON JAVA S2I FOR OPENSHIFT 16
4.3.1. Prepare for Deployment 16
4.3.2. Deployment 16
4.3.3. Post Deployment 17

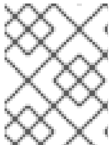
CHAPTER 5. REFERENCE 18
5.1. VERSION DETAILS 18
5.2. INFORMATION ENVIRONMENT VARIABLES 18
5.3. CONFIGURATION ENVIRONMENT VARIABLES 19
5.4. EXPOSED PORTS 24
5.5. CONFIGURING MAVEN SETTINGS 25
5.5.1. Default Maven Settings with Maven Arguments 25
5.5.2. Providing Custom Maven Settings 25

CHAPTER 1. INTRODUCTION

1.1. WHAT IS RED HAT JAVA S2I FOR OPENSHIFT

OpenShift Container Platform provides an S2I (Source-to-Image) process to build and run applications where one can attach an application's source code on top of a builder image (a technology image such as JBoss EAP). S2I process builds your application first and then layers it on top of the builder image to create an application image. After the build is complete, the application image is pushed to the Integrated registry inside OpenShift or to a standalone registry.

Red Hat Java S2I for OpenShift is a Source-to-Image (S2I) builder image designed for use with OpenShift. It allows users to build and run plain Java applications (fat-jar and flat classpath) within a containerized image on OpenShift.



NOTE

The Red Hat Java S2I for OpenShift image is only supported on OpenShift Container Platforms 3.6 and 3.5.

CHAPTER 2. BEFORE YOU BEGIN

2.1. INITIAL SETUP

The instructions in this guide follow on from and assume an OpenShift instance similar to that created in the OpenShift Primer.

2.2. VERSION COMPATIBILITY AND SUPPORT

See the xPaaS part of the OpenShift and Atomic Platform Tested Integrations page for details about OpenShift image version compatibility.

CHAPTER 3. GET STARTED

This section describes some of the ways you can use the Java S2I for OpenShift image to run your custom java applications on OpenShift.

3.1. SOURCE TO IMAGE (S2I) BUILD

To run and configure the Java S2I for OpenShift image, use the OpenShift S2I process.

The S2I process for the Java S2I for OpenShift image works as follows:

1. Log into the OpenShift instance by running the following command and providing credentials.

```
$ oc login
```

2. Create a new project.

```
$ oc new-project <project-name>
```

3. Create a new application using the Java S2I for OpenShift image. *<source-location>* can be the URL of a git repository or a path to a local folder.

```
$ oc new-app redhat-openjdk18-openshift~<source-location>
```

4. Get the service name.

```
$ oc get service
```

5. Expose the service as a route to be able to use it from the browser. *<service-name>* is the value of **`NAME`** field from previous command output.

```
$ oc expose svc/<service-name> --port=8080
```

6. Get the route.

```
$ oc get route
```

7. Access the application in your browser using the URL (value of **`HOST/PORT`** field from previous command output).

3.2. BINARY BUILDS

To deploy existing applications on OpenShift, you can use the binary source capability.

Prerequisite:

A. **Get the JAR application archive or build the application locally.**
The example below uses the undertow-servlet quickstart.

- Clone the source code.

```
$ git clone https://github.com/jboss-openshift/openshift-quickstarts.git
```

- Configure the Red Hat JBoss Middleware Maven repository.
- Build the application.

```
$ cd openshift-quickstarts/undertow-servlet/
```

```
$ mvn clean package
[INFO] Scanning for projects...
...
[INFO]
[INFO] -------------------------------------------------------------------------
[INFO] Building Undertow Servlet Example 1.0.0.Final
[INFO] -------------------------------------------------------------------------
...
[INFO] -------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] -------------------------------------------------------------------------
[INFO] Total time: 1.986 s
[INFO] Finished at: 2017-06-27T16:43:07+02:00
[INFO] Final Memory: 19M/281M
[INFO] -------------------------------------------------------------------------
```

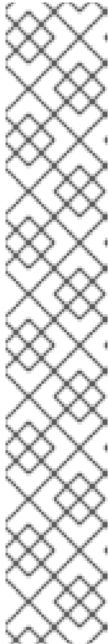
B. **Prepare the directory structure on the local file system.**
Application archives in the **deployments**/ subdirectory of the main binary build directory are copied directly to the standard deployments folder of the image being built on OpenShift. For the application to deploy, the directory hierarchy containing the web application data must be correctly structured.

Create main directory for the binary build on the local file system and **deployments**/ subdirectory within it. Copy the previously built JAR archive to the **deployments**/ subdirectory:

```
undertow-servlet]$ ls
dependency-reduced-pom.xml  pom.xml  README  src  target
```

```
$ mkdir -p ocp/deployments
```

```
$ cp target/undertow-servlet.jar ocp/deployments/
```



NOTE

Location of the standard deployments directory depends on the underlying base image, that was used to deploy the application. See the following table:

Table 3.1. Standard Location of the Deployments Directory

Name of the Underlying Base Image(s)	Standard Location of the Deployments Directory
EAP for OpenShift 6.4 and 7.0	***$JBOSS_HOME/standalone/deployments***
Java S2I for OpenShift	***/deployments***
JWS for OpenShift	***$JWS_HOME/webapps***

Perform the following steps to run application consisting of binary input on OpenShift:

1. Log into the OpenShift instance by running the following command and providing credentials.

```
$ oc login
```

2. Create a new project.

```
$ oc new-project jdk-bin-demo
```

3. (Optional) Identify the image stream for the particular image.

```
$ oc get is -n openshift | grep ^redhat-openjdk | cut -f1 -d ' '
redhat-openjdk18-openshift
```

4. Create new binary build, specifying image stream and application name.

```
$ oc new-build --binary=true \
--name=jdk-us-app \
--image-stream=redhat-openjdk18-openshift
--> Found image c1f5b31 (2 months old) in image stream
"openshift/redhat-openjdk18-openshift" under tag "latest" for
"redhat-openjdk18-openshift"

    Java Applications
    -----------------
    Platform for building and running plain Java applications (fat-
jar and flat classpath)

    Tags: builder, java

    * A source build using binary input will be created
      * The resulting image will be pushed to image stream "jdk-us-
app:latest"
      * A binary build was created, use 'start-build --from-dir' to
trigger a new build
```

```
--> Creating resources with label build=jdk-us-app ...
    imagestream "jdk-us-app" created
    buildconfig "jdk-us-app" created
--> Success
```

5. Start the binary build. Instruct **oc** executable to use main directory of the binary build we created in previous step as the directory containing binary input for the OpenShift build.

```
$ oc start-build jdk-us-app --from-dir=./ocp --follow
Uploading directory "ocp" as binary input for the build ...
build "jdk-us-app-1" started
Receiving source from STDIN as archive ...
==================================================================
Starting S2I Java Build .....
S2I source build with plain binaries detected
Copying binaries from /tmp/src/deployments to /deployments ...
... done
Pushing image 172.30.197.203:5000/jdk-bin-demo/jdk-us-app:latest ...
Pushed 0/6 layers, 2% complete
Pushed 1/6 layers, 24% complete
Pushed 2/6 layers, 36% complete
Pushed 3/6 layers, 54% complete
Pushed 4/6 layers, 71% complete
Pushed 5/6 layers, 95% complete
Pushed 6/6 layers, 100% complete
Push successful
```

6. Create a new OpenShift application based on the build.

```
$ oc new-app jdk-us-app
--> Found image 66f4e0b (About a minute old) in image stream "jdk-
bin-demo/jdk-us-app" under tag "latest" for "jdk-us-app"

    jdk-bin-demo/jdk-us-app-1:c1dbfb7a
    ----------------------------------
    Platform for building and running plain Java applications (fat-
jar and flat classpath)

    Tags: builder, java

    * This image will be deployed in deployment config "jdk-us-app"
    * Ports 8080/tcp, 8443/tcp, 8778/tcp will be load balanced by
service "jdk-us-app"
      * Other containers can access this service through the
hostname "jdk-us-app"

--> Creating resources ...
    deploymentconfig "jdk-us-app" created
    service "jdk-us-app" created
--> Success
    Run 'oc status' to view your app.
```

7. Expose the service as route.

```
$ oc get svc -o name
service/jdk-us-app
```

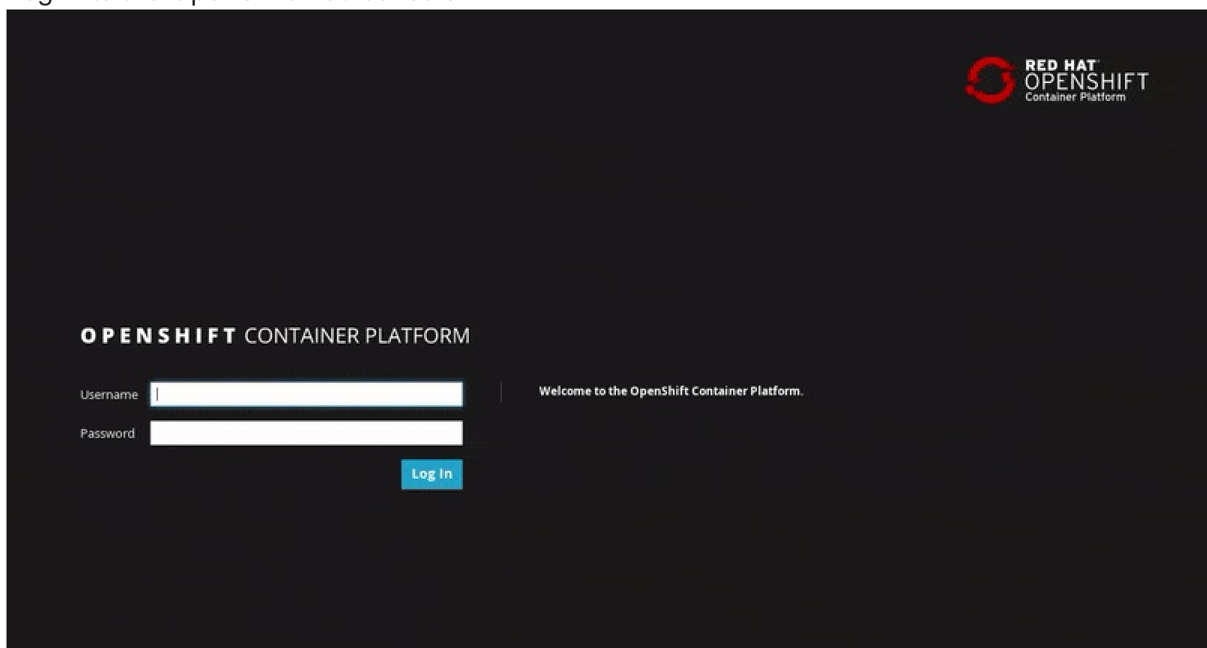
```
$ oc expose svc/jdk-us-app
route "jdk-us-app" exposed
```

8. Access the application.
   Access the application in your browser using the URL **http://jdk-us-app-jdk-bin-demo.openshift.example.com/**.

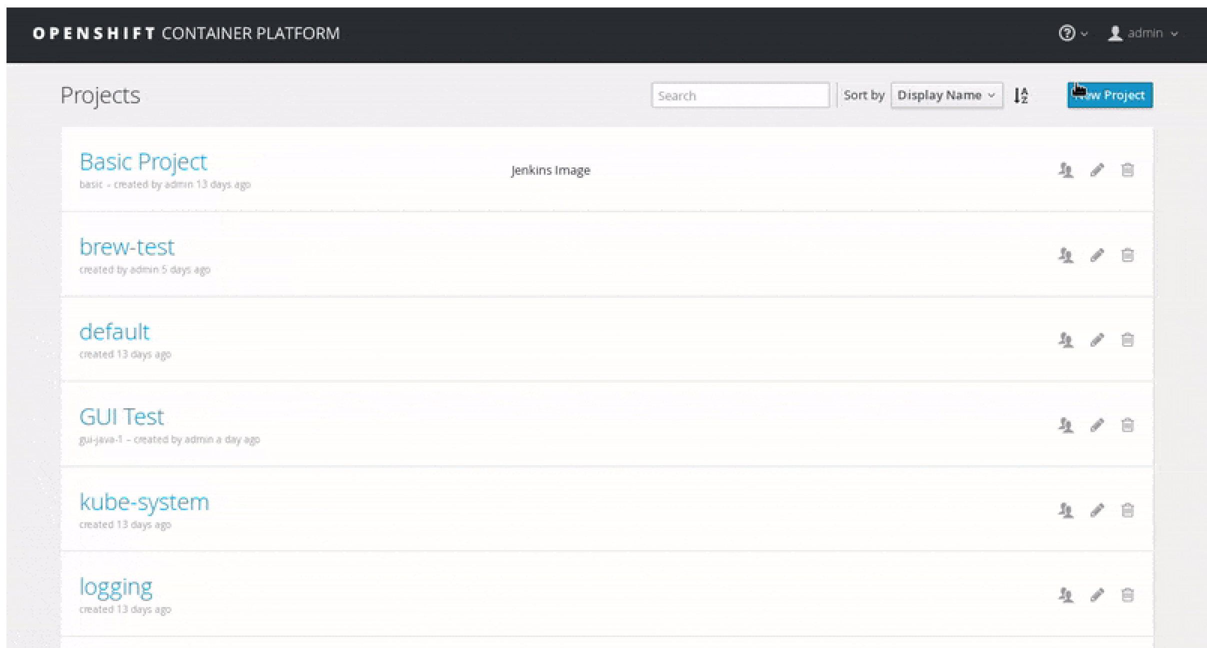
3.3. BUILD USING THE WEB CONSOLE

Configure and deploy your java application using an application template from the OpenShift web console.

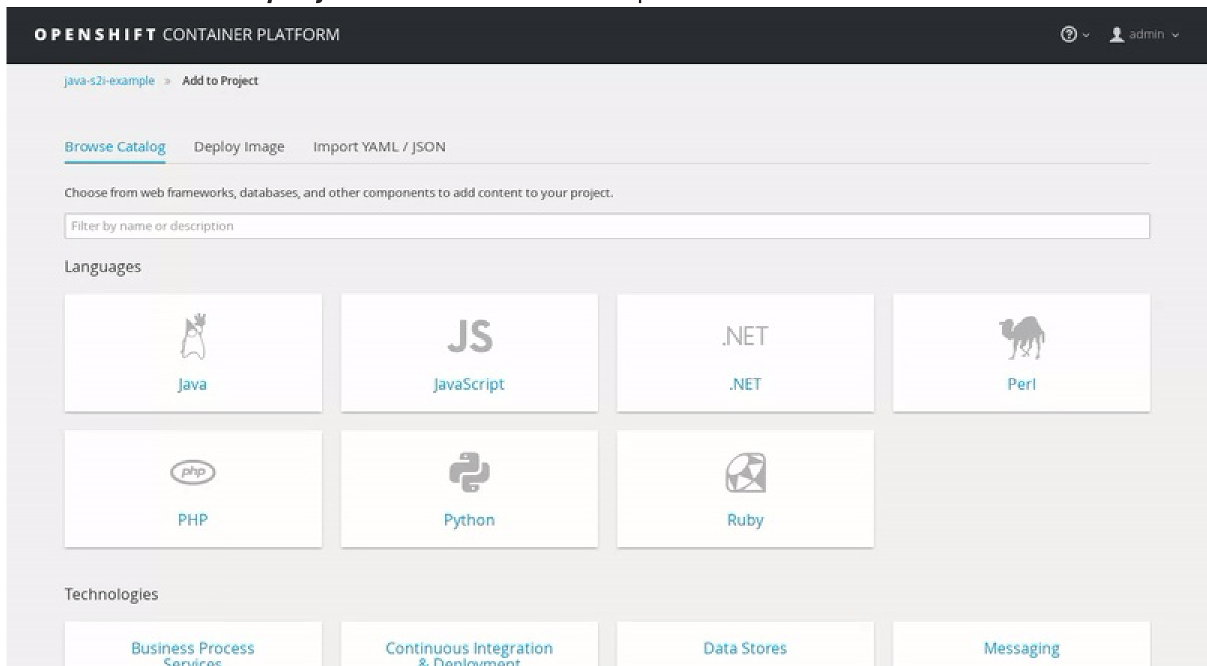
1. Log in to the OpenShift web console.



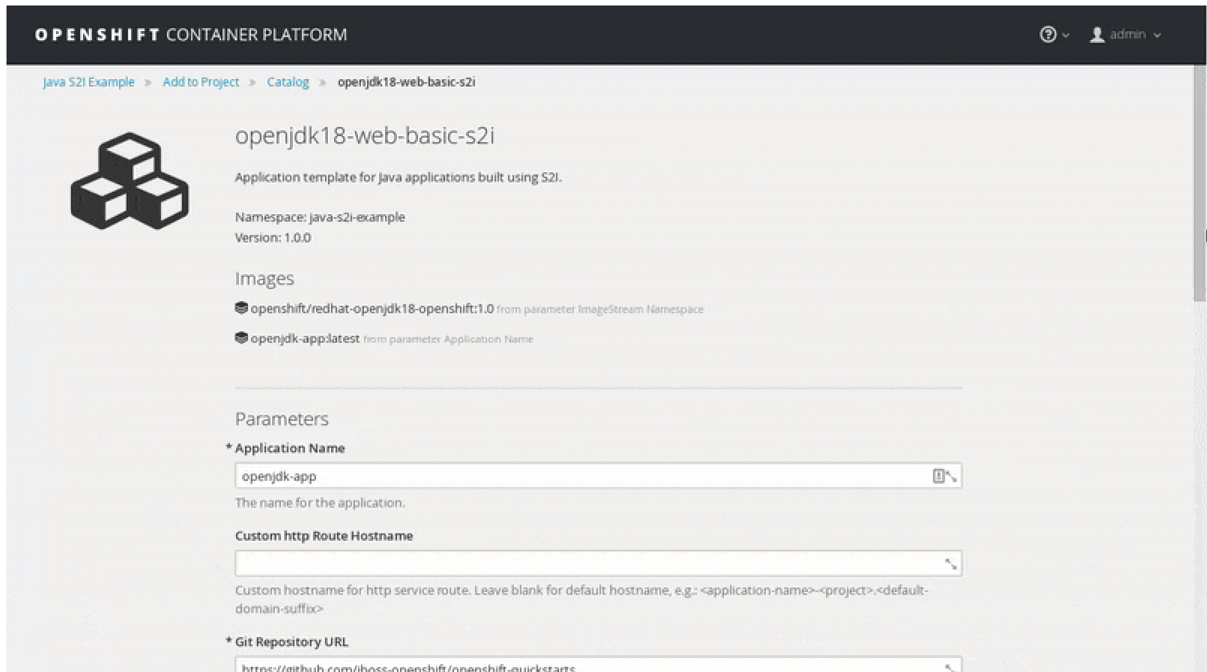
2. Click on **New Project**, enter the details for ***Name***, ***Display Name*** and ***Description*** fields, and then click **Create**.



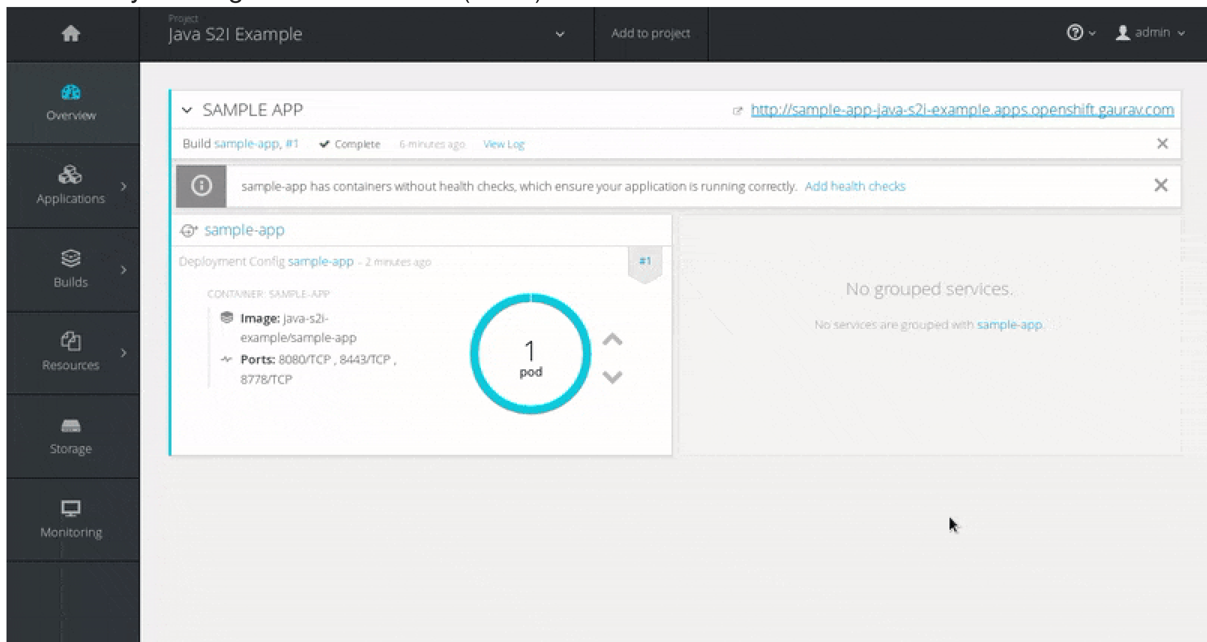
3. Click on the ***Filter by name or description*** text field and type `jdk` to list matching templates. Click **Select** on the ***openjdk18-web-basic-s2i*** template.



4. Leave the default values, scroll to the bottom of the page and click **Create**. Then click **Continue to Overview**.



5. Wait for the build to finish. Once the application pod is running, access the application in your browser by clicking on the listed link (route).



CHAPTER 4. TUTORIALS

4.1. EXAMPLE WORKFLOW: USING MAVEN TO BUILD AND RUN UBER JAR ON JAVA S2I FOR OPENSHIFT IMAGE

This tutorial focuses on building and running Maven applications on OpenShift using the Java S2I for OpenShift image.

4.1.1. Prepare for Deployment

1. Log in to the OpenShift instance by running following command and providing credentials.

   ```
   $ oc login
   ```

2. Create a new project.

   ```
   $ oc new-project js2i-demo
   ```

3. Create a service account to be used for this deployment.

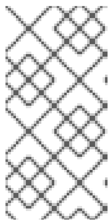
   ```
   $ oc create serviceaccount js2i-service-account
   ```

4. Add the view role to the service account. This enables the service account to view all the resources in the js2i-demo namespace, which is necessary for managing the cluster.

   ```
   $ oc policy add-role-to-user view system:serviceaccount:js2i-demo:js2i-service-account
   ```

5. Generate a self-signed certificate keystore. This example uses ‘keytool’, a package included with the Java Development Kit, to generate dummy credentials for use with the keystore:

   ```
   $ keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -validity 360 -keysize 2048
   ```



NOTE

OpenShift does not permit login authentication from self-signed certificates. For demonstration purposes, this example uses ‘openssl‘ to generate a CA certificate to sign the SSL keystore and create a truststore. This truststore is also included in the creation of the secret, and specified in the SSO template.

WARNING

For production environments, its recommended that you use your own SSL certificate purchased from a verified Certificate Authority (CA) for SSL-encrypted connections (HTTPS).

6. Use the generated keystore file to create the secret.

```
$ oc secrets new js2i-app-secret keystore.jks
```

7. Add the secret to the service account created earlier.

```
$ oc secrets link js2i-service-account js2i-app-secret
```

4.1.2. Deployment

1. Create a new application using the Java S2I for OpenShift image and Java source code.

```
$ oc new-app redhat-openjdk18-openshift~https://github.com/jboss-openshift/openshift-quickstarts.git --context-dir=undertow-servlet
```

2. View the Maven build logs for the example repository by running the following command:

```
$ oc logs -f bc/openshift-quickstarts
```

4.1.3. Post-Deployment

4.1.3.1. Creating a Route

After deployment is finished create a route for the application so that clients outside of OpenShift can connect using SSL.

1. Create a route.

```
$ oc create route edge --service=openshift-quickstarts
```

2. Get route.

```
$ oc get route
```

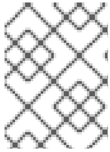
3. Access the application in your browser using the URL (value of **`HOST/PORT`** field from previous command output).

4. Optionally, you can also scale up the application instance by running the following command:

```
$ oc scale dc js2i-demo --replicas=3
```

4.2. EXAMPLE WORKFLOW: REMOTE DEBUGGING A JAVA APPLICATION RUNNING ON JAVA S2I FOR OPENSHIFT IMAGE

This tutorial describes remote debugging of a Java application deployed on OpenShift using the Java S2I for OpenShift image. The capability can be enabled by setting the value of the environment variables **`JAVA_DEBUG`** to **`true`** and **`JAVA_DEBUG_PORT`** to **`9009`**, respectively.



NOTE

If the **JAVA_DEBUG** variable is set to **true** and no value is provided for the **JAVA_DEBUG_PORT** variable, **JAVA_DEBUG_PORT** is set to **5005** by default.

4.2.1. Prepare for Deployment

1. Log in to the OpenShift instance by running following command and providing credentials.

```
$ oc login
```

2. Create a new project:

```
$ oc new-project js2i-remote-debug-demo
```

4.2.2. Deployment

4.2.2.1. Enabling Remote Debugging for a New Application

1. Create a new application using the Java S2I for OpenShift image and example Java source code. Ensure that **JAVA_DEBUG** and **JAVA_DEBUG_PORT** environment variables are set properly when creating the application.

```
$ oc new-app redhat-openjdk18-openshift~https://github.com/jboss-
openshift/openshift-quickstarts.git \
  --context-dir=undertow-servlet \
  -e JAVA_DEBUG=true \
  -e JAVA_DEBUG_PORT=9009
```

Proceed to Connect local debugging port to a port on the pod.

4.2.2.2. Enabling Remote Debugging for an Existing Application

1. Switch to the appropriate OpenShift project.

```
$ oc project js2i-remote-debug-demo
```

2. Retrieve the name of the deployment config.

```
$ oc get dc -o name
deploymentconfig/openshift-quickstarts
```

3. Edit the deployment config with the proper setting of **JAVA_DEBUG** and **JAVA_DEBUG_PORT** variables.

```
$ oc env dc/openshift-quickstarts -e JAVA_DEBUG=true -e
JAVA_DEBUG_PORT=9009
```

Proceed to Connect local debugging port to a port on the pod.

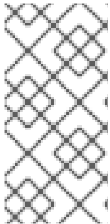
4.2.2.3. Connect Local Debugging Port to a Port on the Pod

1. Get the name of the pod running the application.

```
$ oc get pods
NAME                               READY     STATUS      RESTARTS
AGE
openshift-quickstarts-1-1uymm     1/1       Running     0          3m
openshift-quickstarts-1-build     0/1       Completed   0          6m
```

2. Use the OpenShift / Kubernetes port forwarding feature to listen on a local port and forward to a port on the OpenShift pod.

```
$ oc port-forward openshift-quickstarts-1-1uymm 5005:9009
Forwarding from 127.0.0.1:5005 -> 9009
Forwarding from [::1]:5005 -> 9009
```



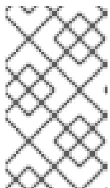
NOTE

In the preceding example, **5005** is the port number on the local system, while **9009** is the remote port number of the OpenShift pod running the Java S2I for OpenShift image. Therefore, future debugging connections made to local port **5005** are forwarded to port **9009** of the OpenShift pod, running the Java Virtual Machine (JVM).

4.2.3. Post-Deployment

1. Attach the debugger on the local system to the remote JVM running on the Java S2I for OpenShift image using the following command:

```
$ jdb -attach 5005
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
>
...
```



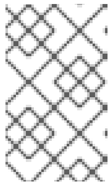
NOTE

Once the local debugger to the remote OpenShift pod debugging connection is initiated, an entry similar to **Handling connection for 5005** is shown in the console where the previous **oc port-forward** command was issued.

2. Debug the application.

```
$ jdb -attach 5005
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> threads
Group system:
  (java.lang.ref.Reference$ReferenceHandler)0x79e Reference Handler
cond. waiting
  (java.lang.ref.Finalizer$FinalizerThread)0x79f  Finalizer
cond. waiting
  (java.lang.Thread)0x7a0                         Signal Dispatcher
```

```
running
Group main:
  (java.util.TimerThread)0x7a2                    server-timer
cond. waiting
  (org.jolokia.jvmagent.CleanupThread)0x7a3       Jolokia Agent
Cleanup Thread cond. waiting
  (org.xnio.nio.WorkerThread)0x7a4                XNIO-1 I/O-1
running
  (org.xnio.nio.WorkerThread)0x7a5                XNIO-1 I/O-2
running
  (org.xnio.nio.WorkerThread)0x7a6                XNIO-1 I/O-3
running
  (org.xnio.nio.WorkerThread)0x7a7                XNIO-1 Accept
running
  (java.lang.Thread)0x7a8                         DestroyJavaVM
running
Group jolokia:
  (java.lang.Thread)0x7aa                         Thread-3
running
>
```



NOTE

For more information on connecting the IDE debugger of the Red Hat JBoss Developer Studio to the OpenShift pod running the Java S2I for OpenShift image, refer to Configuring and Connecting the IDE Debugger.

4.3. EXAMPLE WORKFLOW: RUNNING FLAT CLASSPATH JAR ON JAVA S2I FOR OPENSHIFT

This tutorial describes the process of running flat classpath java applications on Java S2I for OpenShift.

4.3.1. Prepare for Deployment

1. Log in to the OpenShift instance by running following command and providing credentials.

```
$ oc login
```

2. Create a new project.

```
$ oc new-project js2i-flatclasspath-demo
```

4.3.2. Deployment

1. Create a new application using the Java S2I for OpenShift image and Java source code.

```
$ oc new-app redhat-openjdk18-openshift~https://github.com/jboss-
openshift/openshift-quickstarts.git --context-dir=undertow-servlet
```

2. Retrieve the name of the build config.

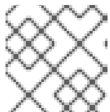
```
$ oc get bc -o name
buildconfig/openshift-quickstarts
```

3. Edit the build config by specifying values for the **JAVA_MAIN_CLASS**, **MAVEN_ARGS**, **ARTIFACT_COPY_ARGS**, **JAVA_LIB_DIR**, **JAVA_APP_JAR**, and **JAVA_APP_DIR** environment variables.

```
$ oc env bc/openshift-quickstarts \
-e
JAVA_MAIN_CLASS=org.openshift.quickstarts.undertow.servlet.ServletSe
rver \
-e MAVEN_ARGS="package -P flat-classpath-jar -
Dcom.redhat.xpaas.repo.redhatga" \
-e ARTIFACT_COPY_ARGS="-r lib *.jar" \
-e JAVA_LIB_DIR=lib \
-e JAVA_APP_JAR=undertow-servlet.jar \
-e JAVA_APP_DIR=/deployments
```

4. Rebuild the application using the updated build config.

```
$ oc start-build openshift-quickstarts --follow
```



NOTE

The **--follow** tag retrieves the build logs and shows them in the console.

4.3.3. Post Deployment

1. Get the service name.

```
$ oc get service
```

2. Expose the service as a route to be able to use it from the browser.

```
$ oc expose svc/openshift-quickstarts --port=8080
```

3. Get the route.

```
$ oc get route
```

4. Access the application in your browser using the URL (value of **HOST/PORT** field from previous command output).

CHAPTER 5. REFERENCE

5.1. VERSION DETAILS

The table below lists versions of technologies used in this image.

Table 5.1. Technologies used and their version

Technology	Version
OpenJDK	8
Jolokia	1.3.5
Maven	3.3.9-2.8

5.2. INFORMATION ENVIRONMENT VARIABLES

The following information environment variables are designed to convey information about the image and should not be modified by the user:

Table 5.2. Information Environment Variables

Variable Name	Description	Example Value
HOME	-	***/home/jboss***
JAVA_DATA_DIR	-	***/deployments/data***
JAVA_HOME	-	***/usr/lib/jvm/java-1.8.0***
JAVA_VENDOR	-	***openjdk***
JAVA_VERSION	-	***1.8.0***
JBOSS_IMAGE_NAME	Image name, same as Name label	***redhat-openjdk-18/openjdk18-openshift***
JBOSS_IMAGE_RELEASE	Image release, same as Release label.	***2***
JBOSS_IMAGE_VERSION	Image version, same as Version label.	***1.0***
JOLOKIA_VERSION	-	***1.3.5***
MAVEN_VERSION	-	***3.3.9-2.8.el7***

Variable Name	Description	Example Value
PATH	-	***$PATH:"/usr/local/s2i"***

5.3. CONFIGURATION ENVIRONMENT VARIABLES

Configuration environment variables are designed to conveniently adjust the image without requiring a rebuild, and should be set by the user as desired.

Table 5.3. Configuration Environment Variables

Variable Name	Description	Example Value
AB_JOLOKIA_AUTH_OPENSHIFT	Switch on client authentication for OpenShift TLS communication. The value of this parameter can be a relative distinguished name which must be contained in a presented client certificate. Enabling this parameter will automatically switch Jolokia into https communication mode. The default CA cert is set to **`/var/run/secrets/kubernetes.io/serviceaccount/ca.crt`**	***true***
AB_JOLOKIA_CONFIG	If set uses this file (including path) as Jolokia JVM agent properties (as described in Jolokia's reference manual). If not set, the ***/opt/jolokia/etc/jolokia.properties*** file will be created using the settings as defined in this document, otherwise the rest of the settings in this document are ignored.	***/opt/jolokia/custom.properties***
AB_JOLOKIA_DISCOVERY_ENABLED	Enable Jolokia discovery. Defaults to ***false***.	***true***
AB_JOLOKIA_HOST	Host address to bind to, the default address is **`0.0.0.0`**.	***127.0.0.1***

Variable Name	Description	Example Value
AB_JOLOKIA_HTTPS	Switch on secure communication with https. By default self-signed server certificates are generated if no serverCert configuration is given in ***AB_JOLOKIA_OPTS***. *NOTE: If the values is set to an empty string, https is turned* **`off`**. *If the value is set to a non empty string, https is turned* **`on`**.	***true***
AB_JOLOKIA_OFF	If set disables activation of Jolokia (i.e. echos an empty value). By default, Jolokia is enabled. *NOTE: If the values is set to an empty string, https is turned* **`off`**. *If the value is set to a non empty string, https is turned* **`on`**.	***true***
AB_JOLOKIA_OPTS	Additional options to be appended to the agent configuration. They should be given in the format **`"key=value, key=value, … "`**	***backlog=20***
AB_JOLOKIA_PASSWORD	Password for basic authentication. By default authentication is switched off.	***mypassword***
AB_JOLOKIA_PASSWORD_RANDOM	If set, a random value is generated for ***AB_JOLOKIA_PASSWORD***, and it is saved in the ***/opt/jolokia/etc/jolokia.pw*** file.	***true***
AB_JOLOKIA_PORT	Port to use (Default: ***8778***)	***5432***
AB_JOLOKIA_USER	User for basic authentication. Defaults to 'jolokia'	***myusername***

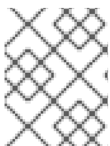
Variable Name	Description	Example Value
ARTIFACT_COPY_ARGS	Arguments to use when copying artifacts from the output directory to the application directory. Useful to specify which artifacts will be part of the image. It defaults to `-r hawt-app/` when a hawt-app directory is found on the build directory, otherwise jar files only will be included (.jar).	*-r hawt-app/**
ARTIFACT_DIR	Path to target/ where the jar files are created for multi-module builds. These are added to *MAVEN_ARGS*	*/plugins*
CONTAINER_CORE_LIMIT	A calculated core limit as described in CFS Bandwidth Control	*2*
CONTAINER_MAX_MEMORY	Memory limit given to the container. This value must be in bytes.	*536870912* (which results into -Xmx256 (default ratio is 50%))
GC_ADAPTIVE_SIZE_POLICY_WEIGHT	The weighting given to the current Garbage Collection (GC) time versus previous GC times.	*90*
GC_MAX_HEAP_FREE_RATIO	Maximum percentage of heap free after GC to avoid shrinking.	*40*
GC_MAX_METASPACE_SIZE	The maximum metaspace size.	*100*
GC_MIN_HEAP_FREE_RATIO	Minimum percentage of heap free after GC to avoid expansion.	*20*
GC_TIME_RATIO	Specifies the ratio of the time spent outside the garbage collection (for example, the time spent for application execution) to the time spent in the garbage collection.	*4*
HTTP_PROXY	The location of the http proxy, this will be used for both Maven builds and Java runtime	*127.0.0.1:8080*

Variable Name	Description	Example Value
http_proxy	The location of the http proxy, this takes precedence over ***HTTP_PROXY*** and will be used for both Maven builds and Java runtime	***http://127.0.0.1:8080***
HTTPS_PROXY	The location of the https proxy, this takes precedence over ***http_proxy*** and ***HTTP_PROXY*** and will be used for both Maven builds and Java runtime	***myuser@127.0.0.1:8080***
https_proxy	The location of the https proxy, this takes precedence over ***http_proxy***, ***HTTP_PROXY***, and ***HTTPS_PROXY*** and will be used for both Maven builds and Java runtime	***myuser:mypass@127.0.0.1:8080***
JAVA_APP_DIR	The directory where the application resides. All paths in your application are relative to this directory.	***myapplication/***
JAVA_APP_JAR	A jar file with an appropriate manifest so that it can be started with Java **`-jar`** if no ***JAVA_MAIN_CLASS*** is set. In all cases this jar file is added to the classpath, too.	Configuration dependent.†
JAVA_APP_NAME	Name to use for the process	***demo-app***
JAVA_ARGS	Arguments passed to the Java application	***hello_world***
JAVA_CLASSPATH	The classpath to use. If ***JAVA_LIB_DIR*** is set, the startup script checks for a file ***JAVA_LIB_DIR/classpath***. If it is not set, the startup script checks for a file ***JAVA_APP_DIR/classpath*** and use its content as classpath. If this file doesn't exists all jars in the application directory are added (***classes:JAVA_APP_DIR/****).	Configuration dependent.†

Variable Name	Description	Example Value
JAVA_DEBUG	If set remote debugging will be switched on	***true***
JAVA_DEBUG_PORT	Port used for remote debugging. Default: ***5005***	***9009***
JAVA_DIAGNOSTICS	Set this to get some diagnostics information to standard out when things are happening	***true***
JAVA_LIB_DIR	Directory holding the Java jar files as well an optional classpath file which holds the classpath. Either as a single-line classpath (colon separated) or with jar files listed line-by-line. If not set ***JAVA_LIB_DIR*** is the same as ***JAVA_APP_DIR***.	Configuration dependent.†
JAVA_MAIN_CLASS	A main class to use as argument for Java. When this environment variable is given, all jar files in ***JAVA_APP_DIR*** are added to the classpath as well as ***JAVA_LIB_DIR***.	***com.example.MyMainClass***
JAVA_MAX_MEM_RATIO	It is used when no -Xmx option is given in ***JAVA_OPTIONS***. This is used to calculate a default maximal Heap Memory based on a containers restriction. If used in a Docker container without any memory constraints for the container then this option has no effect. If there is a memory constraint then **-Xmx** is set to a ratio of the container available memory as set here. The default is 50 which means 50% of the available memory is used as an upper boundary. You can skip this mechanism by setting this value to 0 in which case no **-Xmx** option is added.	***40***
JAVA_OPTIONS	JVM options passed to the Java command	***-verbose:class***

Variable Name	Description	Example Value
MAVEN_ARGS	Arguments to use when calling Maven, replacing the default value **`-e -Popenshift -DskipTests -Dcom.redhat.xpaas.repo.redhatga -Dfabric8.skip=true package`**. Also read Default Maven settings with Maven Arguments	*-e -Popenshift -DskipTests -Dcom.redhat.xpaas.repo.redhatga package*
MAVEN_ARGS_APPEND	Additional Maven arguments	*-X -am -pl*
MAVEN_CLEAR_REPO	If set then the Maven repository is removed after the artifact is built. This is useful for keeping the created application image small, but prevents incremental builds. The default is false	*true*
MAVEN_MIRROR_URL	The base URL of a mirror used for retrieving artifacts	*http://10.0.0.1:8080/repository/internal/*
NO_PROXY	A comma-separated lists of hosts, IP addresses or domains that can be accessed directly, this will be used for both Maven builds and Java runtime	*foo.example.com,bar.example.com*
no_proxy	A comma-separated lists of hosts, IP addresses or domains that can be accessed directly, this takes precedence over ***NO_PROXY*** and will be used for both Maven builds and Java runtime	**.example.com*

† *Varies depending on the configuration, therefore no generic example is provided.*



NOTE

Other environment variables not listed above that can influence the product can be found in JBOSS documentation.

5.4. EXPOSED PORTS

Port Number	Description
8080	HTTP

Port Number	Description
8443	HTTPS
8778	Jolokia Monitoring

5.5. CONFIGURING MAVEN SETTINGS

5.5.1. Default Maven Settings with Maven Arguments

The default value of **`MAVEN_ARGS`** environment variable contains the **`-Dcom.redhat.xpaas.repo.redhatga`** property. This property activates a profile with the ***https://maven.repository.redhat.com/ga/*** repository within the default **`jboss-settings.xml`** file, which resides in the Java S2I for OpenShift image.

When specifying a custom value for the **`MAVEN_ARGS`** environment variable, if a custom **`source_dir/configuration/settings.xml`** file is not specified, the default **`jboss-settings.xml`** in the image is used.

To specify which Maven repository will be used within the default **`jboss-settings.xml`**, there are two properties:

1. The **`-Dcom.redhat.xpaas.repo.redhatga`** property, to use the ***https://maven.repository.redhat.com/ga/*** repository.
2. The **`-Dcom.redhat.xpaas.repo.jbossorg`** property to use the ***https://repository.jboss.org/nexus/content/groups/public/*** repository.

5.5.2. Providing Custom Maven Settings

To specify a custom **`settings.xml`** file along with Maven arguments, create the **`source_dir/configuration`** directory and place the **`settings.xml`** file inside.

Sample path should be similar to: **`source_dir/configuration/settings.xml`**.