Red Hat JBoss Enterprise Application Platform 7.3

Using Eclipse MicroProfile with JBoss EAP XP 2.0.0

For Use with JBoss EAP XP 2.0.0

Last Updated: 2021-01-11

Red Hat JBoss Enterprise Application Platform 7.3 Using Eclipse MicroProfile with JBoss EAP XP 2.0.0

For Use with JBoss EAP XP 2.0.0

Legal Notice

Copyright © 2021 Red Hat, Inc.

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

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

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

Linux ® 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 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

This document provides general information about using Eclipse MicroProfile in JBoss EAP XP 2.0.0.

Table of Contents

CHAPTER 1. JBOSS EAP XP FOR THE LATEST MICROPROFILE CAPABILITIES 6
1.1. ABOUT JBOSS EAP XP 6
1.2. JBOSS EAP XP INSTALLATION 6
1.3. JBOSS EAP XP MANAGER FOR MANAGING JBOSS EAP XP PATCH STREAMS 6
1.4. JBOSS EAP XP MANAGER 2.0 COMMANDS 7
1.5. INSTALLING JBOSS EAP XP 2.0.0 ON JBOSS EAP 7.3.X 9
1.6. UPGRADING JBOSS EAP XP 1.0.X TO 2.0.0 10
1.7. UNINSTALLING JBOSS EAP XP 12
1.8. VIEWING THE STATUS OF JBOSS EAP XP 12

CHAPTER 2. UNDERSTAND ECLIPSE MICROPROFILE 14
2.1. ECLIPSE MICROPROFILE CONFIG 14
2.1.1. Eclipse MicroProfile Config in JBoss EAP 14
2.1.2. Eclipse MicroProfile Config sources supported in Eclipse MicroProfile Config 14
2.2. ECLIPSE MICROPROFILE FAULT TOLERANCE 15
2.2.1. About Eclipse MicroProfile Fault Tolerance specification 15
2.2.2. Eclipse MicroProfile Fault Tolerance in JBoss EAP 15
2.3. ECLIPSE MICROPROFILE HEALTH 16
2.3.1. Eclipse MicroProfile Health in JBoss EAP 16
2.4. ECLIPSE MICROPROFILE JWT 17
2.4.1. Eclipse MicroProfile JWT integration in JBoss EAP 17
2.4.2. Differences between a traditional deployment and an Eclipse MicroProfile JWT deployment 17
2.4.3. Eclipse MicroProfile JWT activation in JBoss EAP 17
2.4.4. Limitations of Eclipse MicroProfile JWT in JBoss EAP 18
2.5. ECLIPSE MICROPROFILE METRICS 18
2.5.1. Eclipse MicroProfile Metrics in JBoss EAP 18
2.6. ECLIPSE MICROPROFILE OPENAPI 19
2.6.1. Eclipse MicroProfile OpenAPI in JBoss EAP 19
2.7. ECLIPSE MICROPROFILE OPENTRACING 19
2.7.1. Eclipse MicroProfile OpenTracing 19
2.7.2. Eclipse MicroProfile OpenTracing in EAP 19
2.8. ECLIPSE MICROPROFILE REST CLIENT 20
2.8.1. MicroProfile REST client 20

CHAPTER 3. ADMINISTER ECLIPSE MICROPROFILE IN JBOSS EAP 22
3.1. ECLIPSE MICROPROFILE OPENTRACING ADMINISTRATION 22
3.1.1. Enabling MicroProfile Open Tracing 22
3.1.2. Removing the microprofile-opentracing-smallrye subsystem 22
3.1.3. Adding the microprofile-opentracing-smallrye subsystem 22
3.1.4. Installing Jaeger 23
3.2. ECLIPSE MICROPROFILE CONFIG CONFIGURATION 23
3.2.1. Adding properties in a ConfigSource management resource 23
3.2.2. Configuring directories as ConfigSources 23
3.2.3. Obtaining ConfigSource from a ConfigSource class 24
3.2.4. Obtaining ConfigSource configuration from a ConfigSourceProvider class 24
3.3. ECLIPSE MICROPROFILE FAULT TOLERANCE CONFIGURATION 25
3.3.1. Adding the MicroProfile Fault Tolerance extension 25
3.4. ECLIPSE MICROPROFILE HEALTH CONFIGURATION 26
3.4.1. Examining health using the management CLI 26
3.4.2. Examining health using the management console 26
3.4.3. Examining health using the HTTP endpoint 26

3.4.4. Enabling authentication for Eclipse MicroProfile Health 27
3.4.5. Readiness probes that determine server health and readiness 27
3.4.6. Global status when probes are not defined 28
3.5. ECLIPSE MICROPROFILE JWT CONFIGURATION 29
3.5.1. Enabling microprofile-jwt-smallrye subsystem 29
3.6. ECLIPSE MICROPROFILE METRICS ADMINISTRATION 29
3.6.1. Metrics available on the management interface 29
3.6.2. Examining metrics using the HTTP endpoint 30
3.6.3. Enabling Authentication for the Eclipse MicroProfile Metrics HTTP Endpoint 30
3.6.4. Obtaining the request count for a web service 30
3.7. ECLIPSE MICROPROFILE OPENAPI ADMINISTRATION 31
3.7.1. Enabling Eclipse MicroProfile OpenAPI 31
3.7.2. Requesting an Eclipse MicroProfile OpenAPI document using Accept HTTP header 32
3.7.3. Requesting an Eclipse MicroProfile OpenAPI document using an HTTP parameter 32
3.7.4. Configuring JBoss EAP to serve a static OpenAPI document 33
3.7.5. Disabling microprofile-openapi-smallrye 34
3.8. STANDALONE SERVER CONFIGURATION 34
3.8.1. Standalone server configuration files 34
3.8.2. Updating standalone configurations with Eclipse MicroProfile subsystems and extensions 35

CHAPTER 4. DEVELOP ECLIPSE MICROPROFILE APPLICATIONS FOR JBOSS EAP 37
4.1. MAVEN AND THE JBOSS EAP ECLIPSE MICROPROFILE MAVEN REPOSITORY 37
4.1.1. Downloading the JBoss EAP Eclipse MicroProfile Maven repository patch as an archive file 37
4.1.2. Applying the JBoss EAP Eclipse MicroProfile Maven repository patch on your local system 37
4.1.3. Supported JBoss EAP Eclipse MicroProfile BOM 38
4.1.4. Using the JBoss EAP Eclipse MicroProfile Maven repository 39
4.2. ECLIPSE MICROPROFILE CONFIG DEVELOPMENT 40
4.2.1. Creating a Maven project for Eclipse MicroProfile Config 40
4.2.2. Using MicroProfile Config property in an application 41
4.3. ECLIPSE MICROPROFILE FAULT TOLERANCE APPLICATION DEVELOPMENT 43
4.3.1. Adding the MicroProfile Fault Tolerance extension 43
4.3.2. Configuring Maven project for Eclipse MicroProfile Fault Tolerance 44
4.3.3. Creating a fault tolerant application 45
4.4. ECLIPSE MICROPROFILE HEALTH DEVELOPMENT 48
4.4.1. Custom health check example 48
4.4.2. The @Liveness annotation example 49
4.4.3. The @Readiness annotation example 49
4.5. ECLIPSE MICROPROFILE JWT APPLICATION DEVELOPMENT 50
4.5.1. Enabling microprofile-jwt-smallrye subsystem 50
4.5.2. Configuring Maven project for developing JWT applications 50
4.5.3. Creating an application with Eclipse MicroProfile JWT 51
4.6. ECLIPSE MICROPROFILE METRICS DEVELOPMENT 56
4.6.1. Creating an Eclipse MicroProfile Metrics application 56
4.7. DEVELOPING AN ECLIPSE MICROPROFILE OPENAPI APPLICATION 58
4.7.1. Enabling Eclipse MicroProfile OpenAPI 58
4.7.2. Configuring Maven project for Eclipse MicroProfile OpenAPI 58
4.7.3. Creating an Eclipse MicroProfile OpenAPI application 60
4.7.4. Configuring JBoss EAP to serve a static OpenAPI document 64
4.8. ECLIPSE MICROPROFILE REST CLIENT DEVELOPMENT 65
4.8.1. A comparison between MicroProfile REST client and JAX-RS syntaxes 65
4.8.2. Programmatic registration of providers in MicroProfile REST client 66
4.8.3. Declarative registration of providers in MicroProfile REST client 66
4.8.4. Declarative specification of headers in MicroProfile REST client 66

4.8.5. Propagation of headers on the server in MicroProfile REST client 67
4.8.6. ResponseExceptionMapper in MicroProfile REST client 68
4.8.7. Context dependency injection with MicroProfile REST client 68

CHAPTER 5. BUILD AND RUN MICROSERVICES APPLICATIONS ON THE OPENSHIFT IMAGE FOR JBOSS EAP XP 70
5.1. PREPARING OPENSHIFT FOR APPLICATION DEPLOYMENT 70
5.2. CONFIGURING AUTHENTICATION TO THE RED HAT CONTAINER REGISTRY 71
5.3. IMPORTING THE LATEST OPENSHIFT IMAGESTREAMS AND TEMPLATES FOR JBOSS EAP XP 71
5.4. DEPLOYING A JBOSS EAP XP SOURCE-TO-IMAGE (S2I) APPLICATION ON OPENSHIFT 73
5.5. COMPLETING POST-DEPLOYMENT TASKS FOR JBOSS EAP XP SOURCE-TO-IMAGE (S2I) APPLICATION 75

CHAPTER 6. CAPABILITY TRIMMING 77
6.1. PROVISION A CUSTOM JBOSS EAP SERVER 77
6.1.1. Available JBoss EAP layers 77
6.1.2. Base layers 77
datasources-web-server 77
jaxrs-server 78
cloud-server 78
6.1.3. Decorator layers 79
ejb-lite 79
ejb 79
ejb-local-cache 79
ejb-dist-cache 80
jdr 80
jpa 80
jpa-distributed 80
jsf 81
microprofile-platform 81
observability 81
remote-activemq 81
sso 82
web-console 82
web-clustering 82
webservices 82
6.2. PROVISIONING USER-DEVELOPED LAYERS IN JBOSS EAP 82
6.2.1. Building custom layers for JBoss EAP 82
6.2.2. Custom provisioning files for JBoss EAP 84
6.2.3. Building an application provisioned with user-developed layers 85

CHAPTER 7. ENABLE ECLIPSE MICROPROFILE APPLICATION DEVELOPMENT FOR JBOSS EAP ON RED HAT CODEREADY STUDIO 87
7.1. CONFIGURING CODEREADY STUDIO TO USE ECLIPSE MICROPROFILE CAPABILITIES 87
7.2. USING ECLIPSE MICROPROFILE QUICKSTARTS FOR CODEREADY STUDIO 88

CHAPTER 8. THE BOOTABLE JAR 90
8.1. ABOUT THE BOOTABLE JAR 90
8.2. JBOSS EAP MAVEN PLUG-IN 90
8.3. BOOTABLE JAR ARGUMENTS 91
8.4. SPECIFYING GALLEON LAYERS FOR YOUR BOOTABLE JAR SERVER 93
8.5. USING A BOOTABLE JAR ON A JBOSS EAP BARE-METAL PLATFORM 95
8.6. CREATING A HOLLOW BOOTABLE JAR ON A JBOSS EAP BARE-METAL PLATFORM 97
8.7. CLI SCRIPTS 98

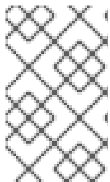
8.8. USING A BOOTABLE JAR ON A JBOSS EAP OPENSHIFT PLATFORM 100
8.9. CONFIGURE THE BOOTABLE JAR FOR OPENSHIFT 102
8.10. USING A CONFIGMAP IN YOUR APPLICATION ON OPENSHIFT 103
8.11. ENABLE WEB-SESSION DATA STORAGE FOR MULTIPLE BOOTABLE JAR INSTANCES 105
8.12. ENABLING HTTP AUTHENTICATION FOR BOOTABLE JAR WITH A CLI SCRIPT 111
8.13. SECURING YOUR JBOSS EAP BOOTABLE JAR APPLICATION WITH RED HAT SINGLE SIGN-ON 115
8.14. PACKAGING A BOOTABLE JAR IN DEV MODE 121
8.15. APPLYING THE JBOSS EAP PATCH TO YOUR BOOTABLE JAR 122

CHAPTER 9. REFERENCE 124
9.1. ECLIPSE MICROPROFILE CONFIG REFERENCE 124
9.1.1. Default Eclipse MicroProfile Config attributes 124
9.1.2. Eclipse MicroProfile Config SmallRye ConfigSources 124
9.2. ECLIPSE MICROPROFILE FAULT TOLERANCE REFERENCE 124
9.2.1. Eclipse MicroProfile Fault Tolerance configuration properties 124
9.3. ECLIPSE MICROPROFILE JWT REFERENCE 125
9.3.1. Eclipse MicroProfile Config JWT standard properties 125
9.4. ECLIPSE MICROPROFILE OPENAPI REFERENCE 125
9.4.1. Eclipse MicroProfile OpenAPI configuration properties 125

CHAPTER 1. JBOSS EAP XP FOR THE LATEST MICROPROFILE CAPABILITIES

1.1. ABOUT JBOSS EAP XP

The Eclipse MicroProfile Expansion Pack (JBoss EAP XP) is available as a patch stream, which is provided using JBoss EAP XP manager.



NOTE

JBoss EAP XP is subject to a separate support and life cycle policy. For more details, see the JBoss Enterprise Application Platform expansion pack Support and Life Cycle Policies page.

The JBoss EAP XP patch provides the following Eclipse MicroProfile 3.3 components:

- Eclipse MicroProfile Config
- Eclipse MicroProfile Fault Tolerance
- Eclipse MicroProfile Health
- Eclipse MicroProfile JWT
- Eclipse MicroProfile Metrics
- Eclipse MicroProfile OpenAPI
- Eclipse MicroProfile OpenTracing
- Eclipse MicroProfile REST Client

1.2. JBOSS EAP XP INSTALLATION

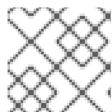
While installing JBoss EAP XP, you must ensure that the version of JBoss EAP XP is compatible with the version of JBoss EAP. JBoss EAP XP 2.0.0 is compatible with JBoss EAP 7.3.4. If you want to install JBoss EAP XP 2.0.0 on JBoss EAP 7.3.0, you must first apply the JBoss EAP 7.3.4 GA patch.

Additional Resources

- Installing JBoss EAP XP 2.0.0 on JBoss EAP 7.3.x

1.3. JBOSS EAP XP MANAGER FOR MANAGING JBOSS EAP XP PATCH STREAMS

JBoss EAP XP manager is an executable **jar** file that you can download from the **Product Downloads** page. Use JBoss EAP XP manager to apply the JBoss EAP XP patches from the JBoss EAP XP patch stream. The patches contain the MicroProfile 3.3 implementations and the bug fixes for these MicroProfile 3.3 implementations.

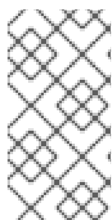


NOTE

You can not manage the JBoss EAP XP patches using the management console.

If you run JBoss EAP XP manager without any arguments, or with the **help** command, you get a list of all the available commands with a description of what they do.

Run the manager with the **help** command to get more information about the arguments available.



NOTE

Most of the JBoss EAP XP manager commands take a **--jboss-home** argument to point to the JBoss EAP XP server to manage the JBoss EAP XP patch stream. Specify the the path to the server in the **JBOSS_HOME** environment variable if you want to omit this. **--jboss-home** takes precedence over the environment variable.

1.4. JBOSS EAP XP MANAGER 2.0 COMMANDS

JBoss EAP XP manager 2.0 provides different commands for managing JBoss EAP XP patch streams.

The following commands are provided:

patch-apply

Use this command to apply patches to your JBoss EAP installation.
The **patch-apply** command is similar to the **patch apply** management CLI command. The **patch-apply** command accepts only those arguments that are required for applying patches using the tool. It uses the default values for other **patch apply** management CLI command arguments.

You can use the **patch-apply** command to apply patches to any patch stream that is enabled on the server. You can also use the command to apply both the base server patches as well as the XP patches.

Example of using the patch-apply command:

```
$ java -jar jboss-eap-xp-manager.jar patch-apply --jboss-home=/PATH/TO/EAP --
patch=/PATH/TO/PATCH/jboss-eap-7.3.4-patch.zip
```

When you apply an XP patch, JBoss EAP XP manager 2.0 performs validation to prevent patch and patch stream mismatch. The following example illustrates incorrect combinations:

- Trying to install JBoss EAP XP 1.0 patch on a server with XP 2.0 patch stream set up causes the following error:

```
java.lang.IllegalStateException: The JBoss EAP XP patch stream in the patch 'jboss-eap-
xp-1.0' does not match the currently enabled JBoss EAP XP patch stream [jboss-eap-xp-
2.0]
 at
org.jboss.eap.util.xp.patch.stream.manager.ManagerPatchApplyAction.doExecute(Manager
PatchApplyAction.java:33)
 at
org.jboss.eap.util.xp.patch.stream.manager.ManagerAction.execute(ManagerAction.java:40)

 at org.jboss.eap.util.xp.patch.stream.manager.ManagerMain.main(ManagerMain.java:50)
```

- Trying to install JBoss EAP XP 2.0 patch on a server that is not set up for JBoss EAP XP 2.0 patch stream causes the following error:

```
java.lang.IllegalStateException: You are attempting to install a patch for the 'jboss-eap-xp-2.0' JBoss EAP XP Patch Stream. However this patch stream is not yet set up in the JBoss EAP server. Run the 'setup' command to enable the patch stream.
 at
org.jboss.eap.util.xp.patch.stream.manager.ManagerPatchApplyAction.doExecute(ManagerPatchApplyAction.java:29)
 at
org.jboss.eap.util.xp.patch.stream.manager.ManagerAction.execute(ManagerAction.java:40)

 at org.jboss.eap.util.xp.patch.stream.manager.ManagerMain.main(ManagerMain.java:50)
```

In both the cases, no changes are made to the server.

remove

Use this command to remove the JBoss EAP XP patch stream setup from the JBoss EAP server.

Example of using the remove command

```
$ java -jar jboss-eap-xp-manager.jar remove --jboss-home=/PATH/TO/EAP
```

setup

Use this command to set up a clean JBoss EAP server for the JBoss EAP XP patch stream. When you use the **setup** command, JBoss EAP XP manager performs the following actions:

- Enables the JBoss EAP XP 2.0 patch stream.
- Applies patches specified using **--base-patch** and **--xp-patch** attributes.
- Copies the **standalone-microprofile.xml** and **standalone-microprofile-ha.xml** configuration files into the server configuration directory.
  If older configuration files are already installed, the new files are saved as timestamped copies in the target configuration directory, such as **standalone-microprofile-yyyyMMdd-HHmmss.xml**.

  You can set the target directory using the **--jboss-config-directory** argument.

Example of using the setup command

```
$ java -jar jboss-eap-xp-manager.jar setup --jboss-home=/PATH/TO/EAP
```

status

Use this command to find the current status of your JBoss EAP XP server. The status command returns the following information:

- The status of the JBoss EAP XP stream.
- Any support policy changes due to being in the current state.
- The major version of JBoss EAP XP.

- Enabled patch streams and their cumulative patch IDs.
- The available JBoss EAP XP manager commands to change the state.

Example of using the status command

```
$ java -jar jboss-eap-xp-manager.jar status --jboss-home=/PATH/TO/EAP
```

upgrade

Use this command to upgrade an old JBoss EAP XP patch stream to the latest patch stream in the JBoss EAP server.
When you use the **upgrade** command, JBoss EAP XP manager performs the following actions:

- Creates a backup of the files enabling the old patch stream in the server.
- Enables the JBoss EAP XP 2.0 patch stream.
- Applies patches specified using **--base-patch** and **--xp-patch** attributes.
- Copies the **standalone-microprofile.xml** and **standalone-microprofile-ha.xml** configuration files into the server configuration directory. If older configuration files are already installed, the new files are saved as timestamped copies in the target configuration directory, such as **standalone-microprofile-yyyyMMdd-HHmmss.xml**.
- If something goes wrong, JBoss EAP XP manager attempts to restore the previous patch stream from the backup it created.
You can set the target directory using the **--jboss-config-directory** argument

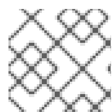
Example of using the upgrade command:

```
$ java -jar jboss-eap-xp-manager.jar upgrade --jboss-home=/PATH/TO/EAP
```

1.5. INSTALLING JBOSS EAP XP 2.0.0 ON JBOSS EAP 7.3.X

Install JBoss EAP XP 2.0.0 on the JBoss EAP 7.3.x base server.

Use JBoss EAP XP 2.0.0 to manage JBoss EAP XP 2.0.0 patch streams.



NOTE

JBoss EAP XP 2.0.0 is certified with JBoss EAP 7.3.4.

Prerequisites

- You have downloaded the following files from the **Product Downloads** page:
  - The **jboss-eap-xp-2.0.0-manager.jar** file (JBoss EAP XP manager 2.0)
  - JBoss EAP 7.3.4 GA patch
  - JBoss EAP XP 2.0.0 patch

Procedure

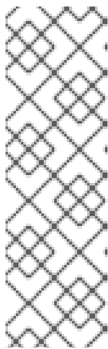
1. If you have not applied the JBoss EAP 7.3.4 GA patch patch to your JBoss EAP server, apply the JBoss EAP 7.3.4 GA patch using the JBoss EAP XP manager 2.0.0 **patch-apply** command:

```
$ java -jar jboss-eap-xp-manager.jar patch-apply --jboss-home=/PATH/TO/EAP --patch=/PATH/TO/PATCH/jboss-eap-7.3.4-patch.zip
```

   The **patch-apply** command is similar to the **patch apply** management CLI command. You can use the **patch apply** management CLI command to apply the patch as well.

2. Set up JBoss EAP XP manager 2.0.0 to manage the EAP XP 2.0 patch stream using the following command:

```
$ java -jar jboss-eap-xp-manager.jar setup --jboss-home=/PATH/TO/EAP
```



   NOTE

   You can apply the JBoss EAP XP 2.0.0 patch at the same time. Include the path to the JBoss EAP XP 2.0.0 patch using the **--xp-patch** argument.

   Example:

```
$ java -jar jboss-eap-xp-manager.jar setup --jboss-home=/PATH/TO/EAP --xp-patch /PATH/TO/PATCH/jboss-eap-xp-2.0.0-patch.zip
```

   The server is now ready to manage the JBoss EAP XP 2.0 patch stream.

3. Optional: If you did not use the the **--xp-patch** argument with the **setup** command, apply the JBoss EAP XP 2.0.0 patch using the JBoss EAP XP manager 2.0.0 **patch-apply** command:

```
$ java -jar jboss-eap-xp-manager.jar patch-apply --jboss-home=/PATH/TO/EAP --patch=/PATH/TO/PATCH/jboss-eap-xp-2.0.0-patch.zip
```

The server is now ready to manage the JBoss EAP XP 2.0 patch stream and is patched with the JBoss EAP XP 2.0.0 patch.

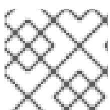
Additional Resources

- JBoss EAP XP manager 2.0 commands

1.6. UPGRADING JBOSS EAP XP 1.0.X TO 2.0.0

Upgrade JBoss EAP XP 1.0.x to 2.0.0 using the **upgrade** command that is provided in JBoss EAP XP manager.

Use JBoss EAP XP 2.0.0 to manage JBoss EAP XP 2.0.0 patch streams.



NOTE

JBoss EAP XP 2.0.0 is certified with JBoss EAP 7.3.4.

Prerequisites

- The base JBoss EAP server is updated to 7.3.4 or later version.

Procedure

1. Verify that your server is in the correct state for upgrade:

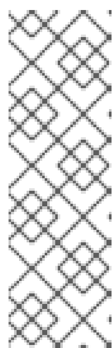
```
$ java -jar jboss-eap-xp-manager.jar status --jboss-home=/PATH/TO/EAP-XP
...
You are currently on JBoss EAP XP 1.
You are using an old version of JBoss EAP XP. The current version is 2, please upgrade.
Enabled patch streams and their cumulative patch ids:
- Patch stream: 'JBoss EAP'; Cumulative patch id: 'jboss-eap-7.3.1.CP'
- Patch stream: 'jboss-eap-xp-1.0'; Cumulative patch id: 'jboss-eap-xp-1.0.0.CP'
Available commands in this state are: [remove, upgrade]
```

   If you see output similar to the above, you can upgrade your JBoss EAP XP server.

   If the output indicates that the server is in an inconsistent state, follow the Troubleshooting steps.

2. Use the **upgrade** command to upgrade JBoss EAP XP 1.0.x to 2.0.0.

```
$ java -jar jboss-eap-xp-manager.jar upgrade --jboss-home=/PATH/TO/EAP
```



   NOTE

   You can apply the JBoss EAP XP 2.0.0 patch at the same time. Include the path to the JBoss EAP XP 2.0.0 patch using the **--xp-patch** argument.

   Example:

```
$ java -jar jboss-eap-xp-manager.jar upgrade --jboss-home=/PATH/TO/EAP --
xp-patch /PATH/TO/PATCH/jboss-eap-xp-2.0.0-patch.zip
```

3. Accept the support policy prompt by entering **yes**.
   The server is now ready to manage the JBoss EAP XP 2.0 patch stream.

4. Optional: If you did not use the the **--xp-patch** argument with the **upgrade** command, use the JBoss EAP XP manager 2.0.0 **patch-apply** command to apply the JBoss EAP XP 2.0.0 patch patch:

```
$ java -jar jboss-eap-xp-manager.jar patch-apply --jboss-home=/PATH/TO/EAP --
patch=/PATH/TO/PATCH/jboss-eap-xp-2.0.0-patch.zip
```

   Use the JBoss EAP XP manager 2.0.0 **patch-apply** command to apply the latest JBoss EAP XP 2.0.x patches when they are available.

Your server is ready to manage the JBoss EAP XP 2.0 patch stream and is patched with the JBoss EAP XP 2.0.0 patch.

Troubleshooting steps

1. Remove the JBoss EAP XP patch stream using the **remove** command of JBoss EAP XP manager 2.0.0:

   ```
   $ java -jar jboss-eap-xp-manager.jar remove --jboss-home=/PATH/TO/EAP
   ```

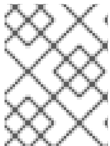
2. Re-install JBoss EAP XP 2.0.0.

Additional resources

- Installing JBoss EAP XP 2.0.0 on JBoss EAP 7.3.x

1.7. UNINSTALLING JBOSS EAP XP

Uninstalling JBoss EAP XP removes all the files related to enabling the JBoss EAP XP 2.0.0 patch stream and the Eclipse MicroProfile 3.3 functionality. The uninstallation process does not affect anything in the base server patch stream or functionality.



NOTE

The uninstallation process does not remove any configuration files, including the ones you added to the JBoss EAP XP patches when you enabled the JBoss EAP XP patch stream.

Procedure

- Uninstall JBoss EAP XP 2.0.0 by issuing the following command:

  ```
  $ java -jar jboss-eap-xp-manager.jar remove --jboss-home=/PATH/TO/EAP
  ```

To install Eclipse MicroProfile 3.3 functionality again, run the **setup** command again to enable the patch stream, and then apply JBoss EAP XP patches to add the Eclipse MicroProfile 3.3. modules.

1.8. VIEWING THE STATUS OF JBOSS EAP XP

You can view the following information with the **status** command:

- The status of the JBoss EAP XP stream.
- Any support policy changes due to being in the current state.
- The major version of JBoss EAP XP.
- Enabled patch streams and their cumulative patch ids.
- The available JBoss EAP XP manager commands to change the state.

JBoss EAP XP can be in one of the following states:

Not set up

JBoss EAP is clean and does not have JBoss EAP XP set up.

Set up

JBoss EAP has JBoss EAP XP set up. The version of the XP patch stream is not displays as the user can use CLI to determine it.

Inconsistent

The files relating to the JBoss EAP XP are in an inconsistent state. This is an error condition and should not happen normally. If you encounter this error, remove the JBoss EAP XP manager as described in the Uninstalling JBoss EAP XP topic and install JBoss EAP XP again using the **setup** command.

Procedure

- View the status of JBoss EAP XP by issuing the following command:

```
$ java -jar jboss-eap-xp-manager.jar status --jboss-home=/PATH/TO/EAP
```

Additional Resources

- Uninstalling JBoss EAP XP
- Installing JBoss EAP XP 2.0.0 on JBoss EAP 7.3.1

CHAPTER 2. UNDERSTAND ECLIPSE MICROPROFILE

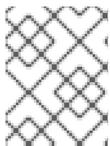
2.1. ECLIPSE MICROPROFILE CONFIG

2.1.1. Eclipse MicroProfile Config in JBoss EAP

Configuration data can change dynamically and applications need to be able to access the latest configuration information without restarting the server.

Eclipse MicroProfile Config provides portable externalization of configuration data. This means, you can configure applications and microservices to run in multiple environments without modification or repackaging.

Eclipse MicroProfile Config functionality is implemented in JBoss EAP using the SmallRye Config component and is provided by the **microprofile-config-smallrye** subsystem. This subsystem is included in the default JBoss EAP 7.3 configuration.



NOTE

Eclipse MicroProfile Config is only supported in JBoss EAP XP. It is not supported in JBoss EAP.

Additional Resources

- Eclipse MicroProfile Config
- SmallRye Config

2.1.2. Eclipse MicroProfile Config sources supported in Eclipse MicroProfile Config

Eclipse MicroProfile Config configuration properties can come from different locations and can be in different formats. These properties are provided by ConfigSources. ConfigSources are implementations of the **org.eclipse.microprofile.config.spi.ConfigSource** interface.

The Eclipse MicroProfile Config specification provides the following default **ConfigSource** implementations for retrieving configuration values:

- **System.getProperties()**.
- **System.getenv()**.
- All **META-INF/microprofile-config.properties** files on the class path.

The **microprofile-config-smallrye** subsystem supports additional types of **ConfigSource** resources for retrieving configuration values. You can also retrieve the configuration values from the following resources:

- Properties in a **microprofile-config-smallrye/config-source** management resource
- Files in a directory
- **ConfigSource** class
- **ConfigSourceProvider** class

Additional Resources

- **org.eclipse.microprofile.config.spi.ConfigSource**

2.2. ECLIPSE MICROPROFILE FAULT TOLERANCE

2.2.1. About Eclipse MicroProfile Fault Tolerance specification

The Eclipse MicroProfile Fault Tolerance specification defines strategies to deal with errors inherent in distributed microservices.

The Eclipse MicroProfile Fault Tolerance specification defines the following strategies to handle errors:

Timeout

Define the amount of time within which an execution must finish. Defining a timeout prevents waiting for an execution indefinitely.

Retry

Define the criteria for retrying a failed execution.

Fallback

Provide an alternative in the case of a failed execution.

CircuitBreaker

Define the number of failed execution attempts before temporarily stopping. You can define the length of the delay before resuming execution.

Bulkhead

Isolate failures in part of the system so that the rest of the system can still function.

Asynchronous

Execute client request in a separate thread.

Additional Resources

- Eclipse MicroProfile Fault Tolerance specification

2.2.2. Eclipse MicroProfile Fault Tolerance in JBoss EAP

The **microprofile-fault-tolerance-smallrye** subsystem provides support for Eclipse MicroProfile Fault Tolerance in JBoss EAP. The subsystem is available only in the JBoss EAP XP stream.

The **microprofile-fault-tolerance-smallrye** subsystem provides the following annotations for interceptor bindings:

- **@Timeout**
- **@Retry**
- **@Fallback**
- **@CircuitBreaker**
- **@Bulkhead**
- **@Asynchronous**

You can bind these annotations at the class level or at the method level. An annotation bound to a class applies to all of the business methods of that class.

The following rules apply to binding interceptors:

- If a component class declares or inherits a class-level interceptor binding, the following restrictions apply:
  - The class must not be declared final.
  - The class must not contain any static, private, or final methods.
- If a non-static, non-private method of a component class declares a method level interceptor binding, neither the method nor the component class may be declared final.

Fault tolerance operations have the following restrictions:

- Fault tolerance interceptor bindings must be applied to a bean class or bean class method.
- When invoked, the invocation must be the business method invocation as defined in CDI specification.
- An operation is not considered fault tolerant if both of the following conditions are true:
  - The method itself is not bound to any fault tolerance interceptor.
  - The class containing the method is not bound to any fault tolerance interceptor.

The **microprofile-fault-tolerance-smallrye** subsystem provides the following configuration options, in addition to the configuration options provided by Eclipse MicroProfile Fault Tolerance:

- **io.smallrye.faulttolerance.globalThreadPoolSize**
- **io.smallrye.faulttolerance.timeoutExecutorThreads**

Additional Resources

- Eclipse MicroProfile Fault Tolerance Specification
- SmallRye Fault Tolerance project

2.3. ECLIPSE MICROPROFILE HEALTH

2.3.1. Eclipse MicroProfile Health in JBoss EAP

JBoss EAP includes the SmallRye Health component, which you can use to determine whether the JBoss EAP instance is responding as expected. This capability is enabled by default.

Eclipse MicroProfile Health is only available when running JBoss EAP as a standalone server.

The Eclipse MicroProfile Health specification defines the following health checks:

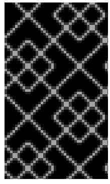
Readiness

Determines whether an application is ready to process requests. The annotation **@Readiness** provides this health check.

Liveness

Determines whether an application is running. The annotation **@Liveness** provides this health check.

The **@Health** annotation defined in previous versions of Eclipse MicroProfile Health specification is deprecated.



IMPORTANT

The **:empty-readiness-checks-status** and the **:empty-liveness-checks-status** management attributes specify the global status when no **readiness** or **liveness** probes are defined.

Additional Resources

- Global status when probes are not defined
- SmallRye Health
- Eclipse MicroProfile Health
- Implement a Custom Health Check

2.4. ECLIPSE MICROPROFILE JWT

2.4.1. Eclipse MicroProfile JWT integration in JBoss EAP

The subsystem **microprofile-jwt-smallrye** provides Eclipse MicroProfile JWT integration in JBoss EAP.

The following functionalities are provided by the **microprofile-jwt-smallrye** subsystem:

- Detecting deployments that use Eclipse MicroProfile JWT security.
- Activating support for Eclipse MicroProfile JWT.

The subsystem contains no configurable attributes or resources.

In addition to the **microprofile-jwt-smallrye** subsystem, the **org.eclipse.microprofile.jwt.auth.api** module provides Eclipse MicroProfile JWT integration in JBoss EAP.

Additional Resources

- SmallRye JWT

2.4.2. Differences between a traditional deployment and an Eclipse MicroProfile JWT deployment

Eclipse MicroProfile JWT deployments do not depend on managed SecurityDomain resources like traditional JBoss EAP deployments. Instead, a virtual SecurityDomain is created and used across the Eclipse MicroProfile JWT deployment.

As the Eclipse MicroProfile JWT deployment is configured entirely within the Eclipse MicroProfile Config properties and the **microprofile-jwt-smallrye** subsystem, the virtual SecurityDomain does not need any other managed configuration for the deployment.

2.4.3. Eclipse MicroProfile JWT activation in JBoss EAP

Eclipse MicroProfile JWT is activated for applications based on the presence of an **auth-method** in the application.

The Eclipse MicroProfile JWT integration is activated for an application in the following way:

- As part of the deployment process, JBoss EAP scans the application archive for the presence of an **auth-method**.
- If an **auth-method** is present and defined as **MP-JWT**, the Eclipse MicroProfile JWT integration is activated.

The **auth-method** can be specified in either or both of the following files:

- the file containing the class that extends **javax.ws.rs.core.Application**, annotated with the **@LoginConfig**
- the **web.xml** configuration file

If **auth-method** is defined both in a class, using annotation, and in the web.xml configuration file, the definition in **web.xml** configuration file is used.

2.4.4. Limitations of Eclipse MicroProfile JWT in JBoss EAP

The Eclipse MicroProfile JWT implementation in JBoss EAP has certain limitations.

The following limitations of Eclipse MicroProfile JWT implementation exist in JBoss EAP:

- The Eclipse MicroProfile JWT implementation parses only the first key from the JSON Web Key Set (JWKS) supplied in the **mp.jwt.verify.publickey** property. Therefore, if a token claims to be signed by the second key or any key after the second key, the token fails verification and the request containing the token is not authorized.
- Base64 encoding of JWKS is not supported.

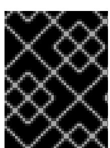
In both cases, a clear text JWKS can be referenced instead of using the **mp.jwt.verify.publickey.location** config property.

2.5. ECLIPSE MICROPROFILE METRICS

2.5.1. Eclipse MicroProfile Metrics in JBoss EAP

JBoss EAP includes the SmallRye Metrics component. The SmallRye Metrics component provides the Eclipse MicroProfile Metrics functionality using the **microprofile-metrics-smallrye** subsystem.

The **microprofile-metrics-smallrye** subsystem provides monitoring data for the JBoss EAP instance. The subsystem is enabled by default.



IMPORTANT

The **microprofile-metrics-smallrye** subsystem is only enabled in standalone configurations.

Additional Resources

- SmallRye Metrics

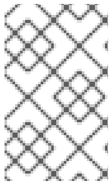
- Eclipse MicroProfile Metrics

2.6. ECLIPSE MICROPROFILE OPENAPI

2.6.1. Eclipse MicroProfile OpenAPI in JBoss EAP

Eclipse MicroProfile OpenAPI is integrated in JBoss EAP using the **microprofile-openapi-smallrye** subsystem.

The Eclipse MicroProfile OpenAPI specification defines an HTTP endpoint that serves an OpenAPI 3.0 document. The OpenAPI 3.0 document describes the REST services for the host. The OpenAPI endpoint is registered using the configured path, for example http://localhost:8080/openapi, local to the root of the host associated with a deployment.



NOTE

Currently, the OpenAPI endpoint for a virtual host can only document a single deployment. To use OpenAPI with multiple deployments registered with different context paths on the same virtual host, each deployment must use a distinct endpoint path.

The OpenAPI endpoint returns a YAML document by default. You can also request a JSON document using an Accept HTTP header, or a format query parameter.

If the Undertow server or host of a given application defines an HTTPS listener then the OpenAPI document is also available using HTTPS. For example, an endpoint for HTTPS is https://localhost:8443/openapi.

2.7. ECLIPSE MICROPROFILE OPENTRACING

2.7.1. Eclipse MicroProfile OpenTracing

The ability to trace requests across service boundaries is important, especially in a microservices environment where a request can flow through multiple services during its life cycle.

The Eclipse MicroProfile OpenTracing specification defines behaviors and an API for accessing an OpenTracing compliant **Tracer** interface within a CDI-bean application. The **Tracer** interface automatically traces JAX-RS applications.

The behaviors specify how OpenTracing Spans are created automatically for incoming and outgoing requests. The API defines how to explicitly disable or enable tracing for given endpoints.

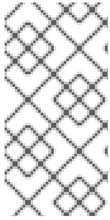
Additional Resources

- For more information about Eclipse MicroProfile OpenTracing specification, see Eclipse MicroProfile OpenTracing documentation.
- For more information about the **Tracer** interface, see **Tracer** javadoc.

2.7.2. Eclipse MicroProfile OpenTracing in EAP

You can use the **microprofile-opentracing-smallrye** subsystem to specify environment variables that trace Jakarta EE applications. This subsystem uses the SmallRye OpenTracing component to provide the Eclipse MicroProfile OpenTracing functionality for JBoss EAP.

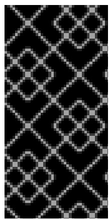
MicroProfile 1.3.0 supports tracing requests for applications. You can configure the default Jaeger Java Client tracer, plus a set of instrumentation libraries for components commonly used in Jakarta EE, to set system properties or environment variables.



NOTE

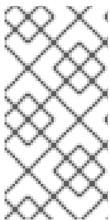
Each individual WAR deployed to the JBoss EAP server automatically has its own **Tracer** instance. Each WAR within an EAR is treated as an individual WAR, and each has its own **Tracer** instance. By default, the service name used with the Jaeger Client is derived from the deployment's name, which is usually the WAR file name.

Within the **microprofile-opentracing-smallrye** subsystem, you can configure the Jaeger Java Client by setting system properties or environment variables.



IMPORTANT

Configuring the Jaeger Client tracer using system properties and environment variables is provided as a Technology Preview. The system properties and environment variables affiliated with the Jaeger Client tracer might change and become incompatible with each other in future releases.



NOTE

By default, the probabilistic sampling strategy of the Jaeger Client for Java is set to **0.001**, meaning that only approximately one in one thousand traces are sampled. To sample every request, set the system properties **JAEGER_SAMPLER_TYPE** to **const** and **JAEGER_SAMPLER_PARAM** to **1**.

Additional Resources

- For more information about SmallRye OpenTracing functionality, see the SmallRye OpenTracing component.
- For more information about the default tracer, see the Jaeger Java Client.
- For more information about the **Tracer** interface, see **Tracer** javadoc.
- For more information about overriding the default tracer and tracing CDI beans, see Using Eclipse MicroProfile OpenTracing to Trace Requests in the *Development Guide*.
- For more information about configuring the Jaeger Client, see the Jaeger documentation.
- For more information about valid system properties, see Configuration via Environment in the Jaeger documentation.

2.8. ECLIPSE MICROPROFILE REST CLIENT

2.8.1. MicroProfile REST client

JBoss EAP XP 2.0.0 supports the MicroProfile REST client 1.4.x that builds on JAX-RS 2.1 client APIs to provide a type-safe approach to invoke RESTful services over HTTP. The MicroProfile Type Safe REST clients are defined as Java interfaces. With the MicroProfile REST clients, you can write client applications with executable code.

Use the MicroProfile REST client to avail the following capabilities:

- An intuitive syntax
- Programmatic registration of providers
- Declarative registration of providers
- Declarative specification of headers
- Propagation of headers on the server
- **ResponseExceptionMapper**
- CDI integration

Additional resources

- A comparison between MicroProfile REST client and JAX-RS syntaxes
- Programmatic registration of providers in MicroProfile REST client
- Declarative registration of providers in MicroProfile REST client
- Declarative specification of headers in MicroProfile REST client
- Propagation of headers on the server in MicroProfile REST client
- ResponseExceptionMapper in MicroProfile REST client
- Context dependency injection with MicroProfile REST client

CHAPTER 3. ADMINISTER ECLIPSE MICROPROFILE IN JBOSS EAP

3.1. ECLIPSE MICROPROFILE OPENTRACING ADMINISTRATION

3.1.1. Enabling MicroProfile Open Tracing

Use the following management CLI commands to enable the MicroProfile Open Tracing feature globally for the server instance by adding the subsystem to the server configuration.

Procedure

1. Enable the **microprofile-opentracing-smallrye** subsystem using the following management command:

```
/subsystem=microprofile-opentracing-smallrye:add()
```

2. Reload the server for the changes to take effect.

```
reload
```

3.1.2. Removing the microprofile-opentracing-smallrye subsystem

The **microprofile-opentracing-smallrye** subsystem is included in the default JBoss EAP 7.3 configuration. This subsystem provides Eclipse MicroProfile OpenTracing functionality for JBoss EAP 7.3. If you experience system memory or performance degradation with MicroProfile OpenTracing enabled, you might want to disable the **microprofile-opentracing-smallrye** subsystem.

You can use the **remove** operation in the management CLI to disable the MicroProfile OpenTracing feature globally for a given server.

Procedure

1. Remove the subsystem.

```
/subsystem=microprofile-opentracing-smallrye:remove()
```

2. Reload the server for the changes to take effect.

```
reload
```

3.1.3. Adding the microprofile-opentracing-smallrye subsystem

You can enable the **microprofile-opentracing-smallrye** subsystem by adding it to the server configuration. Use the **add** operation in the management CLI to enable the MicroProfile OpenTracing feature globally for a given the server.

Procedure

1. Add the subsystem.

```
/subsystem=microprofile-opentracing-smallrye:add()
```

2. Reload the server for the changes to take effect.

```
reload
```

3.1.4. Installing Jaeger

Install Jaeger using **docker**.

Prerequisites

- **docker** is installed.

Procedure

1. Install Jaeger using **docker** by issuing the following command in CLI:

```
$ docker run -d --name jaeger   -p 6831:6831/udp   -p 5778:5778   -p 14268:14268   -p
16686:16686   jaegertracing/all-in-one:1.16
```

3.2. ECLIPSE MICROPROFILE CONFIG CONFIGURATION

3.2.1. Adding properties in a ConfigSource management resource

You can store properties directly in a **config-source** subsystem as a management resource.

Procedure

- Create a ConfigSource and add a property:

```
/subsystem=microprofile-config-smallrye/config-source=props:add(properties={"name" =
"jim"})
```

3.2.2. Configuring directories as ConfigSources

When a property is stored in a directory as a file, the file-name is the name of a property and the file content is the value of the property.

Procedure

1. Create a directory where you want to store the files:

```
$ mkdir -p ~/config/prop-files/
```

2. Navigate to the directory:

```
$ cd ~/config/prop-files/
```

3. Create a file **name** to store the value for the property **name**:

```
$ touch name
```

4. Add the value of the property to the file:

```
$ echo "jim" > name
```

5. Create a ConfigSource in which the file name is the property and the file contents the value of the property:

```
/subsystem=microprofile-config-smallrye/config-source=file-props:add(dir=
{path=~/config/prop-files})
```

This results in the following XML configuration:

```
<subsystem xmlns="urn:wildfly:microprofile-config-smallrye:1.0">
    <config-source name="file-props">
        <dir path="/etc/config/prop-files"/>
    </config-source>
</subsystem>
```

3.2.3. Obtaining ConfigSource from a ConfigSource class

You can create and configure a custom **org.eclipse.microprofile.config.spi.ConfigSource** implementation class to provide a source for the configuration values.

Procedure

- The following management CLI command creates a **ConfigSource** for the implementation class named **org.example.MyConfigSource** that is provided by a JBoss module named **org.example**.
  If you want to use a **ConfigSource** from the **org.example** module, add the **<module name="org.eclipse.microprofile.config.api"/>** dependency to the **path/to/org/example/main/module.xml** file.

```
/subsystem=microprofile-config-smallrye/config-source=my-config-source:add(class=
{name=org.example.MyConfigSource, module=org.example})
```

This command results in the following XML configuration for the **microprofile-config-smallrye** subsystem.

```
<subsystem xmlns="urn:wildfly:microprofile-config-smallrye:1.0">
    <config-source name="my-config-source">
        <class name="org.example.MyConfigSource" module="org.example"/>
    </config-source>
</subsystem>
```

Properties provided by the custom **org.eclipse.microprofile.config.spi.ConfigSource** implementation class are available to any JBoss EAP deployment.

3.2.4. Obtaining ConfigSource configuration from a ConfigSourceProvider class

You can create and configure a custom **org.eclipse.microprofile.config.spi.ConfigSourceProvider** implementation class that registers implementations for multiple **ConfigSource** instances.

Procedure

- Create a **config-source-provider**:

```
/subsystem=microprofile-config-smallrye/config-source-provider=my-config-source-
provider:add(class={name=org.example.MyConfigSourceProvider, module=org.example})
```

  The command creates a **config-source-provider** for the implementation class named **org.example.MyConfigSourceProvider** that is provided by a JBoss Module named **org.example**.

  If you want to use a **config-source-provider** from the **org.example** module, add the **<module name="org.eclipse.microprofile.config.api"/>** dependency to the **path/to/org/example/main/module.xml** file.

  This command results in the following XML configuration for the **microprofile-config-smallrye** subsystem:

```
<subsystem xmlns="urn:wildfly:microprofile-config-smallrye:1.0">
   <config-source-provider name="my-config-source-provider">
      <class name="org.example.MyConfigSourceProvider" module="org.example"/>
   </config-source-provider>
</subsystem>
```

Properties provided by the **ConfigSourceProvider** implementation are available to any JBoss EAP deployment.

Additional resources

- For information about how to add a global module to the JBoss EAP server, see Define Global Modules in the *Configuration Guide* for JBoss EAP.

3.3. ECLIPSE MICROPROFILE FAULT TOLERANCE CONFIGURATION

3.3.1. Adding the MicroProfile Fault Tolerance extension

The MicroProfile Fault Tolerance extension is included in **standalone-microprofile.xml** and **standalone-microprofile-ha.xml** configurations that are provided as part of JBoss EAP XP.

The extension is not included in the standard **standalone.xml** configuration. To use the extension, you must manually enable it.

Prerequisites

- EAP XP pack is installed.

Procedure

1. Add the MicroProfile Fault Tolerance extension using the following management CLI command:

```
/extension=org.wildfly.extension.microprofile.fault-tolerance-smallrye:add
```

2. Enable the **microprofile-fault-tolerance-smallrye** subsystem using the following managenent command:

```
/subsystem=microprofile-fault-tolerance-smallrye:add
```

3. Reload the server with the following management command:

```
reload
```

3.4. ECLIPSE MICROPROFILE HEALTH CONFIGURATION

3.4.1. Examining health using the management CLI

You can check system health using the management CLI.

Procedure

- Examine health:

```
/subsystem=microprofile-health-smallrye:check
{
    "outcome" => "success",
    "result" => {
        "status" => "UP",
        "checks" => []
    }
}
```

3.4.2. Examining health using the management console

You can check system health using the management console.

A check runtime operation shows the health checks and the global outcome as boolean value.

Procedure

1. Navigate to the **Runtime** tab and select the server.
2. In the **Monitor** column, click **MicroProfile Health** → **View**.

3.4.3. Examining health using the HTTP endpoint

Health check is automatically deployed to the health context on JBoss EAP, so you can obtain the current health using the HTTP endpoint.

The default address for the **/health** endpoint, accessible from the management interface, is **http://127.0.0.1:9990/health**.

Procedure

- To obtain the current health of the server using the HTTP endpoint, use the following URL:.

```
http://HOST:PORT/health
```

Accessing this context displays the health check in JSON format, indicating if the server is healthy.

3.4.4. Enabling authentication for Eclipse MicroProfile Health

You can configure the **health** context to require authentication for access.

Procedure

1. Set the **security-enabled** attribute to **true** on the **microprofile-health-smallrye** subsystem.

```
/subsystem=microprofile-health-smallrye:write-attribute(name=security-enabled,value=true)
```

2. Reload the server for the changes to take effect.

```
reload
```

Any subsequent attempt to access the **/health** endpoint triggers an authentication prompt.

3.4.5. Readiness probes that determine server health and readiness

JBoss EAP XP 2.0.0 supports three readiness probes to determine server health and readiness.

- **server-status** - returns **UP** when the server-state is **running**.
- **boot-errors** - returns **UP** when the probe detects no boot errors.
- **deployment-status** - returns **UP** when the status for all deployments is **OK**.

These readiness probes are enabled by default. You can disable the probes using the MicroProfile Config property **mp.health.disable-default-procedures**.

The following example illustrates the use of the three probes with the **check** operation:

```
[standalone@localhost:9990 /] /subsystem=microprofile-health-smallrye:check
{
    "checks": [
        {
            "name": "empty-readiness-checks",
            "status": "UP"
        },
        {
            "name": "empty-liveness-checks",
            "status": "UP"
        },
        {
            "data": {
                "value": "running"
            },
            "name": "server-state",
            "status": "UP"
        },
```

```
        {
          "name": "deployments-status",
          "status": "UP"
        },
        {
          "name": "boot-errors",
          "status": "UP"
        }
      ],
      "status": "UP"
}
```

Additional resources

- Eclipse MicroProfile Health in JBoss EAP
- Global status when probes are not defined

3.4.6. Global status when probes are not defined

The **:empty-readiness-checks-status** and **:empty-liveness-checks-status** management attributes specify the global status when no **readiness** or **liveness** probes are defined.

These attributes allow applications to report 'DOWN' until their probes verify that the application is ready or live. By default, applications report 'UP'.

- The **:empty-readiness-checks-status** attribute specifies the global status for **readiness** probes if no **readiness** probes have been defined:

  ```
  /subsystem=microprofile-health-smallrye:read-attribute(name=empty-readiness-checks-status)
  {
      "outcome" => "success",
      "result" => expression "${env.MP_HEALTH_EMPTY_READINESS_CHECKS_STATUS:UP}"
  }
  ```

- The **:empty-liveness-checks-status** attribute specifies the global status for **liveness** probes if no **liveness** probes have been defined:

  ```
  /subsystem=microprofile-health-smallrye:read-attribute(name=empty-liveness-checks-status)
  {
      "outcome" => "success",
      "result" => expression "${env.MP_HEALTH_EMPTY_LIVENESS_CHECKS_STATUS:UP}"
  }
  ```

  The **/health** HTTP endpoint and the **:check** operation that check both **readiness** and **liveness** probes also take into account these attributes.

You can also modify these attributes as shown in the following example:

```
/subsystem=microprofile-health-smallrye:write-attribute(name=empty-readiness-checks-status,value=DOWN)
{
    "outcome" => "success",
```

```
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

3.5. ECLIPSE MICROPROFILE JWT CONFIGURATION

3.5.1. Enabling microprofile-jwt-smallrye subsystem

The Eclipse MicroProfile JWT integration is provided by the **microprofile-jwt-smallrye** subsystem and is included in the default configuration. If the subsystem is not present in the default configuration, you can add it as follows.

Prerequisites

- EAP XP is installed.

Procedure

1. Enable the MicroProfile JWT smallrye extension in JBoss EAP:

   ```
   /extension=org.wildfly.extension.microprofile.jwt-smallrye:add
   ```

2. Enable the **microprofile-jwt-smallrye** subsystem:

   ```
   /subsystem=microprofile-jwt-smallrye:add
   ```

3. Reload the server:

   ```
   reload
   ```

The **microprofile-jwt-smallrye** subsystem is enabled.

3.6. ECLIPSE MICROPROFILE METRICS ADMINISTRATION

3.6.1. Metrics available on the management interface

The JBoss EAP subsystem metrics are exposed in Prometheus format.

Metrics are automatically available on the JBoss EAP management interface, with the following contexts:

- **/metrics/** - Contains metrics specified in the MicroProfile 3.0 specification.
- **/metrics/vendor** - Contains vendor-specific metrics, such as memory pools.
- **/metrics/application** - Contains metrics from deployed applications and subsystems that use the MicroProfile Metrics API.

The metric names are based on subsystem and attribute names. For example, the subsystem **undertow** exposes a metric attribute **request-count** for every servlet in an application deployment. The name of

this metric is **jboss_undertow_request_count**. The prefix **jboss** identifies JBoss EAP as the source of the metrics.

3.6.2. Examining metrics using the HTTP endpoint

Examine the metrics that are available on the JBoss EAP management interface using the HTTP endpoint.

Procedure

- Use the curl command:

```
$ curl -v http://localhost:9990/metrics | grep -i type
```

3.6.3. Enabling Authentication for the Eclipse MicroProfile Metrics HTTP Endpoint

Configure the **metrics** context to require users to be authorized to access the context. This configuration extends to all the subcontexts of the **metrics** context.

Procedure

1. Set the **security-enabled** attribute to **true** on the **microprofile-metrics-smallrye** subsystem.

```
/subsystem=microprofile-metrics-smallrye:write-attribute(name=security-enabled,value=true)
```

2. Reload the server for the changes to take effect.

```
reload
```

Any subsequent attempt to access the **metrics** endpoint results in an authentication prompt.

3.6.4. Obtaining the request count for a web service

Obtain the request count for a web service that exposes its request count metric.

The following procedure uses **helloworld-rs** quickstart as the web service for obtaining request count. The quickstart is available at Download the quickstart from: jboss-eap-quickstarts.

Prerequsites

- The web service exposes request count.

Procedure

1. Enable statistics for the **undertow** subsystem:

    - Start the standalone server with statistics enabled:

    ```
    $ ./standalone.sh -Dwildfly.statistics-enabled=true
    ```

    - For an already running server, enable the statistics for the **undertow** subsystem:

    ```
    /subsystem=undertow:write-attribute(name=statistics-enabled,value=true)
    ```

2. Deploy the **helloworld-rs** quickstart:
   - In the root directory of the quickstart, deploy the web application using Maven:

     ```
     $ mvn clean install wildfly:deploy
     ```

3. Query the HTTP endpoint in the CLI using the **curl** command and filter for **request_count**:

   ```
   $ curl -v http://localhost:9990/metrics |  grep request_count
   ```

   Expected output:

   ```
   jboss_undertow_request_count_total{server="default-server",http_listener="default",} 0.0
   ```

   The attribute value returned is **0.0**.

4. Access the quickstart, located at http://localhost:8080/helloworld-rs/, in a web browser and click any of the links.

5. Query the HTTP endpoint from the CLI again:

   ```
   $ curl -v http://localhost:9990/metrics |  grep request_count
   ```

   Expected output:

   ```
   jboss_undertow_request_count_total{server="default-server",http_listener="default",} 1.0
   ```

   The value is updated to **1.0**.

   Repeat the last two steps to verify that the request count is updated.

3.7. ECLIPSE MICROPROFILE OPENAPI ADMINISTRATION

3.7.1. Enabling Eclipse MicroProfile OpenAPI

The **microprofile-openapi-smallrye** subsystem is provided in the **standalone-microprofile.xml** configuration. However, JBoss EAP XP uses the **standalone.xml** by default. You must include the subsystem in **standalone.xml** to use it.

Alternatively, you can follow the procedure Updating standalone configurations with Eclipse MicroProfile subsystems and extensions to update the **standalone.xml** configuration file.

Procedure

1. Enable the MicroProfile OpenAPI smallrye extension in JBoss EAP:

   ```
   /extension=org.wildfly.extension.microprofile.openapi-smallrye:add()
   ```

2. Enable the **microprofile-openapi-smallrye** subsystem using the following management command:

   ```
   /subsystem=microprofile-openapi-smallrye:add()
   ```

-

3. Reload the server.

```
reload
```

The **microprofile-openapi-smallrye** subsystem is enabled.

3.7.2. Requesting an Eclipse MicroProfile OpenAPI document using Accept HTTP header

Request an Eclipse MicroProfile OpenAPI document, in the JSON format, from a deployment using an Accept HTTP header.

By default, the OpenAPI endpoint returns a YAML document.

Prerequisites

- The deployment being queried is configured to return an Eclipse MicroProfile OpenAPI document.

Procedure

- Issue the following **curl** command to query the **/openapi** endpoint of the deployment:

```
$ curl -v -H'Accept: application/json' http://localhost:8080/openapi
< HTTP/1.1 200 OK
...
{"openapi": "3.0.1" ... }
```

  Replace http://localhost:8080 with the URL and port of the deployment.

  The Accept header indicates that the JSON document is to be returned using the **application/json** string.

3.7.3. Requesting an Eclipse MicroProfile OpenAPI document using an HTTP parameter

Request an Eclipse MicroProfile OpenAPI document, in the JSON format, from a deployment using a query parameter in an HTTP request.

By default, the OpenAPI endpoint returns a YAML document.

Prerequisites

- The deployment being queried is configured to return an Eclipse MicroProfile OpenAPI document.

Procedure

- Issue the following **curl** command to query the **/openapi** endpoint of the deployment:

```
$ curl -v http://localhost:8080/openapi?format=JSON
< HTTP/1.1 200 OK
...
```

- Replace http://localhost:8080 with the URL and port of the deployment.

  The HTTP parameter **format=JSON** indicates that JSON document is to be returned.

3.7.4. Configuring JBoss EAP to serve a static OpenAPI document

Configure JBoss EAP to serve a static OpenAPI document that describes the REST services for the host.

When JBoss EAP is configured to serve a static OpenAPI document, the static OpenAPI document is processed before any JAX-RS and MicroProfile OpenAPI annotations.

In a production environment, disable annotation processing when serving a static document. Disabling annotation processing ensures that an immutable and versioned API contract is available for clients.

Procedure

1. Create a directory in the application source tree:

   ```
   $ mkdir APPLICATION_ROOT/src/main/webapp/META-INF
   ```

   APPLICATION_ROOT is the directory containing the **pom.xml** configuration file for the application.

2. Query the OpenAPI endpoint, redirecting the output to a file:

   ```
   $ curl http://localhost:8080/openapi?format=JSON > src/main/webapp/META-INF/openapi.json
   ```

   By default, the endpoint serves a YAML document, **format=JSON** specifies that a JSON document is returned.

3. Configure the application to skip annotation scanning when processing the OpenAPI document model:

   ```
   $ echo "mp.openapi.scan.disable=true" > APPLICATION_ROOT/src/main/webapp/META-INF/microprofile-config.properties
   ```

4. Rebuild the application:

   ```
   $ mvn clean install
   ```

5. Deploy the application again using the following management CLI commands:

   a. Undeploy the application:

      ```
      undeploy microprofile-openapi.war
      ```

   b. Deploy the application:

      ```
      deploy APPLICATION_ROOT/target/microprofile-openapi.war
      ```

JBoss EAP now serves a static OpenAPI document at the OpenAPI endpoint.

3.7.5. Disabling microprofile-openapi-smallrye

You can disable the **microprofile-openapi-smallrye** subsystem in JBoss EAP XP using the management CLI.

Procedure

- Disable the **microprofile-openapi-smallrye** subsystem:

```
/subsystem=microprofile-openapi-smallrye:remove()
```

3.8. STANDALONE SERVER CONFIGURATION

3.8.1. Standalone server configuration files

The JBoss EAP XP includes additional standalone server configuration files, **standalone-microprofile.xml** and **standalone-microprofile-ha.xml**.

Standard configuration files that are included with JBoss EAP remain unchanged. Note that JBoss EAP XP 2.0.0 does not support the use of **domain.xml** files or domain mode.

Table 3.1. Standalone configuration files available in JBoss EAP XP

Configuration File	Purpose	Included capabilities	Excluded capabilities
standalone.xml	This is the default configuration that is used when you start your standalone server.	Includes information about the server, including subsystems, networking, deployments, socket bindings, and other configurable details.	Excludes subsystems necessary for messaging or high availability.
standalone-microprofile.xml	This configuration file supports applications that use Eclipse MicroProfile.	Includes information about the server, including subsystems, networking, deployments, socket bindings, and other configurable details.	Excludes the following capabilities: • EJB • Messaging • Java Batch • JavaServer Faces • EJB Timers
standalone-ha.xml		Includes default subsystems and adds the **modcluster** and **jgroups** subsystems for high availability.	Excludes subsystems necessary for messaging.

Configuration File	Purpose	Included capabilities	Excluded capabilities
standalone-microprofile-ha.xml	This standalone file supports applications that use Eclipse MicroProfile.	Includes the **modcluster** and **jgroups** subsystems for high availability in addition to default subsystems.	Excludes subsystems necessary for messaging.
standalone-full.xml		Includes the **messaging-activemq** and **iiop-openjdk** subsystems in addition to default subsystems.	
standalone-full-ha.xml	Support for every possible subsystem.	Includes subsystems for messaging and high availability in addition to default subsystems.	
standalone-load-balancer.xml	Support for the minimum subsystems necessary to use the built-in mod_cluster front-end load balancer to load balance other JBoss EAP instances.		

By default, starting JBoss EAP as a standalone server uses the **standalone.xml** file. To start JBoss EAP with a standalone Eclipse MicroProfile configuration, use the **-c** argument. For example,

```
$ EAP_HOME/bin/standalone.sh -c=standalone-microprofile.xml
```

Additional Resources

- Starting and Stopping JBoss EAP
- Configuration Data

3.8.2. Updating standalone configurations with Eclipse MicroProfile subsystems and extensions

You can update standard standalone server configuration files with Eclipse MicroProfile subsystems and extensions using the **docs/examples/enable-microprofile.cli** script. The **enable-microprofile.cli** script is intended as an example script for updating standard standalone server configuration files, not custom configurations.

The **enable-microprofile.cli** script modifies the existing standalone server configuration and adds the following Eclipse MicroProfile subsystems and extensions if they do not exist in the standalone configuration file:

- **microprofile-openapi-smallrye**

- **microprofile-jwt-smallrye**
- **microprofile-fault-tolerance-smallrye**

The **enable-microprofile.cli** script outputs a high-level description of the modifications. The configuration is secured using the **elytron** subsystem. The **security** subsystem, if present, is removed from the configuration.

Prerequisites

- JBoss EAP XP is installed.

Procedure

1. Run the following CLI script to update the default **standalone.xml** server configuration file:

   ```
   $ EAP_HOME/bin/jboss-cli.sh --file=docs/examples/enable-microprofile.cli
   ```

2. Select a standalone server configuration other than the default **standalone.xml** server configuration file using the following command:

   ```
   $ EAP_HOME/bin/jboss-cli.sh --file=docs/examples/enable-microprofile.cli -Dconfig=
   <standalone-full.xml|standalone-ha.xml|standalone-full-ha.xml>
   ```

3. The specified configuration file now includes Eclipse MicroProfile subsystems and extensions.

CHAPTER 4. DEVELOP ECLIPSE MICROPROFILE APPLICATIONS FOR JBOSS EAP

4.1. MAVEN AND THE JBOSS EAP ECLIPSE MICROPROFILE MAVEN REPOSITORY

4.1.1. Downloading the JBoss EAP Eclipse MicroProfile Maven repository patch as an archive file

Whenever an Eclipse MicroProfile Expansion Pack is released for JBoss EAP, a corresponding patch is provided for the JBoss EAP Eclipse MicroProfile Maven repository. This patch is provided as an incremental archive file that is extracted into the existing Red Hat JBoss Enterprise Application Platform 7.3.0.GA Maven repository. The incremental archive file does not overwrite or remove any existing files, so there is no rollback requirement.

Prerequisites

- You have set up an account on the Red Hat Customer Portal.

Procedure

1. Open a browser and log in to the Red Hat Customer Portal.
2. Select **Downloads** from the menu at the top of the page.
3. Find the **Red Hat JBoss Enterprise Application Platform** entry in the list and select it.
4. From the **Product** drop-down list, select **JBoss EAP XP**.
5. From the **Version** drop-down list, select **2.0.0**.
6. Click the **Releases** tab.
7. Find **JBoss EAP XP 2.0.0 Incremental Maven Repository** in the list, and then click **Download**.
8. Save the archive file to your local directory.

Additional Resources

- To learn more about the JBoss EAP Maven repository, see About the Maven Repository in the JBoss EAP *Development Guide*.

4.1.2. Applying the JBoss EAP Eclipse MicroProfile Maven repository patch on your local system

You can install the JBoss EAP Eclipse MicroProfile Maven repository patch on your local file system.

When you apply a patch in the form of an incremental archive file to the repository, new files are added to this repository. The incremental archive file does not overwrite or remove any existing files on the repository, so there is no rollback requirement.

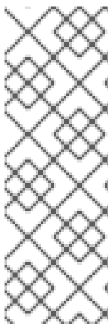
Prerequisites

- You have downloaded and installed the Red Hat JBoss Enterprise Application Platform 7.3.0.GA Maven repository on your local system.
  - Check that you have this minor version of the Red Hat JBoss Enterprise Application Platform 7.3 Maven repository installed on your local system.
- You have downloaded the JBoss EAP XP 2.0.0 Incremental Maven repository on your local system.

Procedure

1. Locate the path to your Red Hat JBoss Enterprise Application Platform 7.3.0.GA Maven repository. For example, **/path/to/repo/jboss-eap-7.3.0.GA-maven-repository/maven-repository/**.
2. Extract the downloaded JBoss EAP XP 2.0.0 Incremental Maven repository directly into the directory of the Red Hat JBoss Enterprise Application Platform 7.3.0.GA Maven repository. For example, open a terminal and issue the following command, replacing the value for your Red Hat JBoss Enterprise Application Platform 7.3.0.GA Maven repository path:

```
$ unzip -o jboss-eap-xp-2.0.0-incremental-maven-repository.zip -d
EAP_MAVEN_REPOSITORY_PATH
```



NOTE

The *EAP_MAVEN_REPOSITORY_PATH* points to the **jboss-eap-7.3.0.GA-maven-repository**. For example, this procedure demonstrated the use of the path **/path/to/repo/jboss-eap-7.3.0.GA-maven-repository/**.

After you extract the JBoss EAP XP Incremental Maven repository into the Red Hat JBoss Enterprise Application Platform 7.3.0.GA Maven repository, the repository name becomes JBoss EAP Eclipse MicroProfile Maven repository.

Additional Resources

- To determine the URL of the JBoss EAP Maven repository, see Determining the URL for the JBoss EAP Maven repository in the JBoss EAP *Development Guide*.

4.1.3. Supported JBoss EAP Eclipse MicroProfile BOM

JBoss EAP XP 2.0.0 includes the JBoss EAP Eclipse MicroProfile BOM. This BOM is named **jboss-eap-xp-microprofile**, and its use case supports JBoss EAP Eclipse MicroProfile APIs.

Table 4.1. JBoss EAP Eclipse MicroProfile BOM

BOM Artifact ID	Use Case
jboss-eap-xp-microprofile	This BOM, whose **groupId** is **org.jboss.bom**, packages many JBoss EAP Eclipse MicroProfile supported API dependencies, such as **microprofile-openapi-api** and **microprofile-config-api**. If you use this BOM, you need not specify a version for a supported API dependency, because the **jboss-eap-xp-microprofile** BOM specifies this value for the dependency.

4.1.4. Using the JBoss EAP Eclipse MicroProfile Maven repository

You can access the **jboss-eap-xp-microprofile** BOM after you install the Red Hat JBoss Enterprise Application Platform 7.3.0.GA Maven repository and apply the JBoss EAP XP Incremental Maven repository to it. The repository name then becomes JBoss EAP Eclipse MicroProfile Maven repository. The BOM is shipped inside the JBoss EAP XP Incremental Maven repository.

You must configure one of the following to use the JBoss EAP Eclipse MicroProfile Maven repository:

- The Maven global or user settings
- The project's POM files

Maven settings used with a repository manager or repository on a shared server provide better control and manageability of projects.

You can use an alternative mirror to redirect all lookup requests for a specific repository to your repository manager without changing the project files.

WARNING

Configuring the JBoss EAP Eclipse MicroProfile Maven repository by modifying the POM file overrides the global and user Maven settings for the configured project.

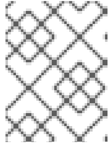
Prerequisites

- You have installed the Red Hat JBoss Enterprise Application Platform 7.3 Maven repository on your local system, and you have applied the JBoss EAP XP Incremental Maven repository to it.

Procedure

1. Choose a configuration method and configure the JBoss EAP Eclipse MicroProfile Maven repository.
2. After you have configured the JBoss EAP Eclipse MicroProfile Maven repository, add the **jboss-eap-xp-microprofile** BOM to the project POM file. The following example shows how to configure the BOM in the **<dependencyManagement>** section of the **pom.xml** file:

```
<dependencyManagement>
  <dependencies>
    ...
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-xp-microprofile</artifactId>
      <version>2.0.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
  </dependency>
    ...
  </dependencies>
</dependencyManagement>
```



NOTE

If you do not specify a value for the **type** element in the **pom.xml** file, Maven specifies a **jar** value for the element.

Additional Resources

- For more information about selecting methods to configure the JBoss EAP Maven repository, see Use the Maven Repository in the JBoss EAP *Development Guide*.
- For more information about managing dependencies, see Dependency Management.

4.2. ECLIPSE MICROPROFILE CONFIG DEVELOPMENT

4.2.1. Creating a Maven project for Eclipse MicroProfile Config

Create a Maven project with the required dependencies and the directory structure for creating an Eclipse MicroProfile Config application.

Prerequisites

- Maven is installed.

Procedure

1. Set up the Maven project.

```
$ mvn archetype:generate \
    -DgroupId=com.example \
    -DartifactId=microprofile-config \
    -DinteractiveMode=false \
    -DarchetypeGroupId=org.apache.maven.archetypes \
    -DarchetypeArtifactId=maven-archetype-webapp
cd microprofile-config
```

   This creates the directory structure for the project and **pom.xml** configuration file.

2. To let the POM file automatically manage the versions for the Eclipse MicroProfile Config artifact and the Eclipse MicroProfile REST Client artifact in the **jboss-eap-xp-microprofile** BOM, import the BOM to the **<dependencyManagement>** section of the project POM file.

```
<dependencyManagement>
  <dependencies>
    <!-- importing the microprofile BOM adds MicroProfile specs -->
    <dependency>
        <groupId>org.jboss.bom</groupId>
        <artifactId>jboss-eap-xp-microprofile</artifactId>
        <version>2.0.0.GA</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

3. Add the Eclipse MicroProfile Config artifact and the Eclipse MicroProfile REST Client artifact and other dependencies, managed by the BOM, to the **<dependency>** section of the project POM file. The following example demonstrates adding the Eclipse MicroProfile Config and the Eclipse MicroProfile REST Client dependencies to the file:

```
<!-- Add the MicroProfile REST Client API. Set provided for the <scope> tag, as the API is included in the server. -->
<dependency>
  <groupId>org.eclipse.microprofile.rest.client</groupId>
  <artifactId>microprofile-rest-client-api</artifactId>
  <scope>provided</scope>
</dependency>
<!-- Add the MicroProfile Config API. Set provided for the <scope> tag, as the API is included in the server. -->
<dependency>
  <groupId>org.eclipse.microprofile.config</groupId>
  <artifactId>microprofile-config-api</artifactId>
  <scope>provided</scope>
</dependency>
<!-- Add the JAX-RS API. Set provided for the <scope> tag, as the API is included in the server. -->
<dependency>
  <groupId>org.jboss.spec.javax.ws.rs</groupId>
  <artifactId>jboss-jaxrs-api_2.1_spec</artifactId>
  <scope>provided</scope>
</dependency>
<!-- Add the CDI API. Set provided for the <scope> tag, as the API is included in the server. -->
<dependency>
  <groupId>jakarta.enterprise</groupId>
  <artifactId>jakarta.enterprise.cdi-api</artifactId>
  <scope>provided</scope>
</dependency>
```

4.2.2. Using MicroProfile Config property in an application

Create an application that uses a configured **ConfigSource**.

Prerequisites

- Eclipse MicroProfile Config is enabled in JBoss EAP.
- The latest POM is installed.
- The Maven project is configured for creating an Eclipse MicroProfile Config application.

Procedure

1. Create the directory to store class files:

```
$ mkdir -p APPLICATION_ROOT/src/main/java/com/example/microprofile/config/
```

Where ***APPLICATION_ROOT*** is the directory containing the **pom.xml** configuration file for the application.

2. Navigate to the new directory:

```
$ cd APPLICATION_ROOT/src/main/java/com/example/microprofile/config/
```

Create all class files described in this procedure in this directory.

3. Create a class file named **HelloApplication.java** with the following content:

```
package com.example.microprofile.config;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/")
public class HelloApplication extends Application {

}
```

This class defines the application as a JAX-RS application.

4. Create a class file named **HelloService.java** with the following content:

```
package com.example.microprofile.config;

public class HelloService {
 String createHelloMessage(String name){
        return "Hello " + name;
    }
}
```

5. Create a class file named **HelloWorld.java** with the following content:

```
package com.example.microprofile.config;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import org.eclipse.microprofile.config.inject.ConfigProperty;

@Path("/config")
public class HelloWorld {

    @Inject
    @ConfigProperty(name="name", defaultValue="jim") (1)
    String name;

    @Inject
    HelloService helloService;

    @GET
    @Path("/json")
    @Produces({ "application/json" })
    public String getHelloWorldJSON() {
        String message = helloService.createHelloMessage(name);
```

```
        return "{\"result\":\"" + message + "\"}";
    }
}
```

1. A MicroProfile Config property is injected into the class with the annotation **@ConfigProperty(name="name", defaultValue="jim")**. If no **ConfigSource** is configured, the value **jim** is returned.

6. Create an empty file named **beans.xml** in the **src/main/webapp/WEB-INF/** directory:

```
$ touch APPLICATION_ROOT/src/main/webapp/WEB-INF/beans.xml
```

   Where ***APPLICATION_ROOT*** is the directory containing the **pom.xml** configuration file for the application.

7. Navigate to the root directory of the application:

```
$ cd APPLICATION_ROOT
```

   Where ***APPLICATION_ROOT*** is the directory containing the **pom.xml** configuration file for the application.

8. Build the project:

```
$ mvn clean install wildfly:deploy
```

9. Test the output:

```
$ curl http://localhost:8080/microprofile-config/config/json
```

   The following is the expected output:

```
{"result":"Hello jim"}
```

4.3. ECLIPSE MICROPROFILE FAULT TOLERANCE APPLICATION DEVELOPMENT

4.3.1. Adding the MicroProfile Fault Tolerance extension

The MicroProfile Fault Tolerance extension is included in **standalone-microprofile.xml** and **standalone-microprofile-ha.xml** configurations that are provided as part of JBoss EAP XP.

The extension is not included in the standard **standalone.xml** configuration. To use the extension, you must manually enable it.

Prerequisites

- EAP XP pack is installed.

Procedure

1. Add the MicroProfile Fault Tolerance extension using the following management CLI command:

```
/extension=org.wildfly.extension.microprofile.fault-tolerance-smallrye:add
```

2. Enable the **microprofile-fault-tolerance-smallrye** subsystem using the following managenent command:

```
/subsystem=microprofile-fault-tolerance-smallrye:add
```

3. Reload the server with the following management command:

```
reload
```

4.3.2. Configuring Maven project for Eclipse MicroProfile Fault Tolerance

Create a Maven project with the required dependencies and the directory structure for creating an Eclipse MicroProfile Fault Tolerance application.

Prerequisites

- Maven is installed.

Procedure

1. Set up the Maven project:

```
mvn archetype:generate \
    -DgroupId=com.example.microprofile.faulttolerance \
    -DartifactId=microprofile-fault-tolerance \
    -DarchetypeGroupId=org.apache.maven.archetypes \
    -DarchetypeArtifactId=maven-archetype-webapp \
    -DinteractiveMode=false
cd microprofile-fault-tolerance
```

The command creates the directory structure for the project and the **pom.xml** configuration file.

2. To let the POM file automatically manage the versions for the Eclipse MicroProfile Fault Tolerance artifact in the **jboss-eap-xp-microprofile** BOM, import the BOM to the **<dependencyManagement>** section of the project POM file.

```
<dependencyManagement>
  <dependencies>
    <!-- importing the microprofile BOM adds MicroProfile specs -->
    <dependency>
        <groupId>org.jboss.bom</groupId>
        <artifactId>jboss-eap-xp-microprofile</artifactId>
        <version>${version.microprofile.bom}</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Replace ${version.microprofile.bom} with the installed version of BOM.

3. Add the Eclipse MicroProfile Fault Tolerance artifact, managed by the BOM, to the **<dependency>** section of the project POM file. The following example demonstrates adding the Eclipse MicroProfile Fault Tolerance dependency to the file:

```
<!-- Add the MicroProfile Fault Tolerance API. Set provided for the <scope> tag, as the API
is included in the server. -->
<dependency>
  <groupId>org.eclipse.microprofile.fault.tolerance</groupId>
  <artifactId>microprofile-fault-tolerance-api</artifactId>
  <scope>provided</scope>
</dependency>
```

4.3.3. Creating a fault tolerant application

Create a fault-tolerant application that implements retry, timeout, and fallback patterns for fault tolerance.

Prerequisites

- Maven dependencies have been configured.

Procedure

1. Create the directory to store class files:

```
$ mkdir -p APPLICATION_ROOT/src/main/java/com/example/microprofile/faulttolerance
```

APPLICATION_ROOT is the directory containing the **pom.xml** configuration file for the application.

2. Navigate to the new directory:

```
$ cd APPLICATION_ROOT/src/main/java/com/example/microprofile/faulttolerance
```

For the following steps, create all class files in the new directory.

3. Create a simple entity representing a coffee sample as **Coffee.java** with the following content:

```
package com.example.microprofile.faulttolerance;

public class Coffee {

    public Integer id;
    public String name;
    public String countryOfOrigin;
    public Integer price;

    public Coffee() {
    }

    public Coffee(Integer id, String name, String countryOfOrigin, Integer price) {
        this.id = id;
        this.name = name;
        this.countryOfOrigin = countryOfOrigin;
```

```
        this.price = price;
    }
}
```

4. Create a class file **CoffeeApplication.java** with the following content:

```
package com.example.microprofile.faulttolerance;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/")
public class CoffeeApplication extends Application {
}
```

5. Create a CDI Bean as **CoffeeRepositoryService.java** with the following content:

```
package com.example.microprofile.faulttolerance;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class CoffeeRepositoryService {

    private Map<Integer, Coffee> coffeeList = new HashMap<>();

    public CoffeeRepositoryService() {
        coffeeList.put(1, new Coffee(1, "Fernandez Espresso", "Colombia", 23));
        coffeeList.put(2, new Coffee(2, "La Scala Whole Beans", "Bolivia", 18));
        coffeeList.put(3, new Coffee(3, "Dak Lak Filter", "Vietnam", 25));
    }

    public List<Coffee> getAllCoffees() {
        return new ArrayList<>(coffeeList.values());
    }

    public Coffee getCoffeeById(Integer id) {
        return coffeeList.get(id);
    }

    public List<Coffee> getRecommendations(Integer id) {
        if (id == null) {
            return Collections.emptyList();
        }
        return coffeeList.values().stream()
                .filter(coffee -> !id.equals(coffee.id))
                .limit(2)
```

```
                .collect(Collectors.toList());
    }
}
```

6. Create a class file **CoffeeResource.java** with the following content:

```
package com.example.microprofile.faulttolerance;

import java.util.List;
import java.util.Random;
import java.util.concurrent.atomic.AtomicLong;
import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.Collections;
import javax.ws.rs.PathParam;
import org.eclipse.microprofile.faulttolerance.Fallback;
import org.eclipse.microprofile.faulttolerance.Timeout;
import org.eclipse.microprofile.faulttolerance.Retry;

@Path("/coffee")
@Produces(MediaType.APPLICATION_JSON)
public class CoffeeResource {

    @Inject
    private CoffeeRepositoryService coffeeRepository;

    private AtomicLong counter = new AtomicLong(0);

    @GET
    @Retry(maxRetries = 4) 1
    public List<Coffee> coffees() {
        final Long invocationNumber = counter.getAndIncrement();
        return coffeeRepository.getAllCoffees();
    }


    @GET
    @Path("/{id}/recommendations")
    @Timeout(250) 2
    public List<Coffee> recommendations(@PathParam("id") int id) {
            return coffeeRepository.getRecommendations(id);
        }

    @GET
    @Path("fallback/{id}/recommendations")
    @Fallback(fallbackMethod = "fallbackRecommendations") 3
    public List<Coffee> recommendations2(@PathParam("id") int id) {
        return coffeeRepository.getRecommendations(id);
        }

    public List<Coffee> fallbackRecommendations(int id) {
        //always return a default coffee
```

```
        return Collections.singletonList(coffeeRepository.getCoffeeById(1));
    }
}
```

1. Define number of re-tries to **4**.
2. Define the timeout interval in milliseconds.
3. Define a fallback method to call when invocation fails.

7. Navigate to the root directory of the application:

```
$ cd APPLICATION_ROOT
```

8. Build the application using the following Maven command:

```
$ mvn clean install wildfly:deploy
```

Access the application at **http://localhost:8080/microprofile-fault-tolerance/coffee**.

Additional Resources

- For a detailed example of fault tolerant application, which includes artificial failures to test the fault tolerance of the application, see the **microprofile-fault-tolerance** quickstart.

4.4. ECLIPSE MICROPROFILE HEALTH DEVELOPMENT

4.4.1. Custom health check example

The default implementation provided by the **microprofile-health-smallrye** subsystem performs a basic health check. For more detailed information, on either the server or application status, custom health checks may be included. Any CDI beans that include the **org.eclipse.microprofile.health.Health** annotation at the class level are automatically discovered and invoked at runtime.

The following example demonstrates how to create a new implementation of a health check that returns an **UP** state.

```
import org.eclipse.microprofile.health.Health;
import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;

@Health
public class HealthTest implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.named("health-test").up().build();
    }
}
```

Once deployed, any subsequent health check queries include the custom checks, as demostrated in the following example.

```
/subsystem=microprofile-health-smallrye:check
{
    "outcome" => "success",
    "result" => {
        "outcome" => "UP",
        "checks" => [{
            "name" => "health-test",
            "state" => "UP"
        }]
    }
}
```

Additional Resources

- https://openliberty.io/javadocs/microprofile-1.2-javadoc/org/eclipse/microprofile/health/Health.html

4.4.2. The @Liveness annotation example

The following is an example of using the **@Liveness** annotation in an application.

```
@Liveness
@ApplicationScoped
public class DataHealthCheck implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.named("Health check with data")
            .up()
            .withData("foo", "fooValue")
            .withData("bar", "barValue")
            .build();
    }
}
```

4.4.3. The @Readiness annotation example

The following example demonstrates checking connection to a database. If the database is down, the readiness check reports error.

```
@Readiness
@ApplicationScoped
public class DatabaseConnectionHealthCheck implements HealthCheck {

    @Inject
    @ConfigProperty(name = "database.up", defaultValue = "false")
    private boolean databaseUp;

    @Override
    public HealthCheckResponse call() {

        HealthCheckResponseBuilder responseBuilder = HealthCheckResponse.named("Database
connection health check");
```

```
        try {
            simulateDatabaseConnectionVerification();
            responseBuilder.up();
        } catch (IllegalStateException e) {
            // cannot access the database
            responseBuilder.down()
                .withData("error", e.getMessage()); // pass the exception message
        }

        return responseBuilder.build();
    }

    private void simulateDatabaseConnectionVerification() {
        if (!databaseUp) {
            throw new IllegalStateException("Cannot contact database");
        }
    }
}
```

4.5. ECLIPSE MICROPROFILE JWT APPLICATION DEVELOPMENT

4.5.1. Enabling microprofile-jwt-smallrye subsystem

The Eclipse MicroProfile JWT integration is provided by the **microprofile-jwt-smallrye** subsystem and is included in the default configuration. If the subsystem is not present in the default configuration, you can add it as follows.

Prerequisites

- EAP XP is installed.

Procedure

1. Enable the MicroProfile JWT smallrye extension in JBoss EAP:

```
/extension=org.wildfly.extension.microprofile.jwt-smallrye:add
```

2. Enable the **microprofile-jwt-smallrye** subsystem:

```
/subsystem=microprofile-jwt-smallrye:add
```

3. Reload the server:

```
reload
```

The **microprofile-jwt-smallrye** subsystem is enabled.

4.5.2. Configuring Maven project for developing JWT applications

Create a Maven project with the required dependencies and the directory structure for developing a JWT application.

Prerequisites

- Maven is installed.
- **microprofile-jwt-smallrye** subsystem is enabled.

Procedure

1. Set up the maven project:

```
$ mvn archetype:generate -DinteractiveMode=false \
    -DarchetypeGroupId=org.apache.maven.archetypes \
    -DarchetypeArtifactId=maven-archetype-webapp \
    -DgroupId=com.example -DartifactId=microprofile-jwt \
    -Dversion=1.0.0.Alpha1-SNAPSHOT
  cd microprofile-jwt
```

   The command creates the directory structure for the project and the **pom.xml** configuration file.

2. To let the POM file automatically manage the versions for the Eclipse MicroProfile JWT artifact in the **jboss-eap-xp-microprofile** BOM, import the BOM to the **<dependencyManagement>** section of the project POM file.

```
<dependencyManagement>
  <dependencies>
    <!-- importing the microprofile BOM adds MicroProfile specs -->
    <dependency>
        <groupId>org.jboss.bom</groupId>
        <artifactId>jboss-eap-xp-microprofile</artifactId>
        <version>${version.microprofile.bom}</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

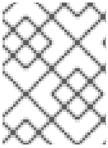
   Replace ${version.microprofile.bom} with the installed version of BOM.

3. Add the Eclipse MicroProfile JWT artifact, managed by the BOM, to the **<dependency>** section of the project POM file. The following example demonstrates adding the Eclipse MicroProfile JWT dependency to the file:

```
<!-- Add the MicroProfile JWT API. Set provided for the <scope> tag, as the API is included in the server. -->
<dependency>
  <groupId>org.eclipse.microprofile.jwt</groupId>
  <artifactId>microprofile-jwt-auth-api</artifactId>
  <scope>provided</scope>
</dependency>
```

4.5.3. Creating an application with Eclipse MicroProfile JWT

Create an application that authenticates requests based on JWT tokens and implements authorization based on the identity of the token bearer.



NOTE

The following procedure provides code for generating tokens as an example. You should implement your own token generator.

Prerequisites

- Maven project is configured with the correct dependencies.

Procedure

1. Create a token generator.
   This step serves as a reference. For a production environment, implement your own token generator.

   a. Create a directory **src/test/java** for token the generator utility and navigate to it:

   ```
   $ mkdir -p src/test/java
   $ cd src/test/java
   ```

   b. Create a class file **TokenUtil.java** with the following content:

   ```
   package  com.example.mpjwt;

   import java.io.FileInputStream;
   import java.io.InputStream;
   import java.nio.charset.StandardCharsets;
   import java.security.KeyFactory;
   import java.security.PrivateKey;
   import java.security.spec.PKCS8EncodedKeySpec;
   import java.util.Base64;
   import java.util.UUID;

   import javax.json.Json;
   import javax.json.JsonArrayBuilder;
   import javax.json.JsonObjectBuilder;

   import com.nimbusds.jose.JOSEObjectType;
   import com.nimbusds.jose.JWSAlgorithm;
   import com.nimbusds.jose.JWSHeader;
   import com.nimbusds.jose.JWSObject;
   import com.nimbusds.jose.JWSSigner;
   import com.nimbusds.jose.Payload;
   import com.nimbusds.jose.crypto.RSASSASigner;

   public class TokenUtil {

       private static PrivateKey loadPrivateKey(final String fileName) throws Exception {
           try (InputStream is = new FileInputStream(fileName)) {
               byte[] contents = new byte[4096];
               int length = is.read(contents);
               String rawKey = new String(contents, 0, length, StandardCharsets.UTF_8)
                       .replaceAll("-----BEGIN (.*)-----", "")
                       .replaceAll("-----END (.*)----", "")
                       .replaceAll("\r\n", "").replaceAll("\n", "").trim();
   ```

```
            PKCS8EncodedKeySpec keySpec = new
PKCS8EncodedKeySpec(Base64.getDecoder().decode(rawKey));
            KeyFactory keyFactory = KeyFactory.getInstance("RSA");

            return keyFactory.generatePrivate(keySpec);
        }
    }

    public static String generateJWT(final String principal, final String birthdate, final
String...groups) throws Exception {
     PrivateKey privateKey = loadPrivateKey("private.pem");

        JWSSigner signer = new RSASSASigner(privateKey);
        JsonArrayBuilder groupsBuilder = Json.createArrayBuilder();
        for (String group : groups) { groupsBuilder.add(group); }

        long currentTime = System.currentTimeMillis() / 1000;
        JsonObjectBuilder claimsBuilder = Json.createObjectBuilder()
                .add("sub", principal)
                .add("upn", principal)
                .add("iss", "quickstart-jwt-issuer")
                .add("aud", "jwt-audience")
                .add("groups", groupsBuilder.build())
                .add("birthdate", birthdate)
                .add("jti", UUID.randomUUID().toString())
                .add("iat", currentTime)
                .add("exp", currentTime + 14400);

        JWSObject jwsObject = new JWSObject(new
JWSHeader.Builder(JWSAlgorithm.RS256)
                .type(new JOSEObjectType("jwt"))
                .keyID("Test Key").build(),
                new Payload(claimsBuilder.build().toString()));

        jwsObject.sign(signer);

        return jwsObject.serialize();
    }

    public static void main(String[] args) throws Exception {
        if (args.length < 2) throw new IllegalArgumentException("Usage TokenUtil {principal}
{birthdate} {groups}");
        String principal = args[0];
        String birthdate = args[1];
        String[] groups = new String[args.length - 2];
        System.arraycopy(args, 2, groups, 0, groups.length);

        String token = generateJWT(principal, birthdate, groups);
        String[] parts = token.split("\\.");
        System.out.println(String.format("\nJWT Header - %s", new
String(Base64.getDecoder().decode(parts[0]), StandardCharsets.UTF_8)));
        System.out.println(String.format("\nJWT Claims - %s", new
String(Base64.getDecoder().decode(parts[1]), StandardCharsets.UTF_8)));
```

```
        System.out.println(String.format("\nGenerated JWT Token \n%s\n", token));
    }
}
```

2. Create the **web.xml** file in the **src/main/webapp/WEB-INF** directory with the following content:

```
<context-param>
    <param-name>resteasy.role.based.security</param-name>
    <param-value>true</param-value>
</context-param>

<security-role>
    <role-name>Subscriber</role-name>
</security-role>
```

3. Create a class file **SampleEndPoint.java** with the following content:

```
package com.example.mpjwt;

import javax.ws.rs.GET;
import javax.ws.rs.Path;

import java.security.Principal;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.SecurityContext;

import javax.annotation.security.RolesAllowed;
import javax.inject.Inject;

import java.time.LocalDate;
import java.time.Period;
import java.util.Optional;

import org.eclipse.microprofile.jwt.Claims;
import org.eclipse.microprofile.jwt.Claim;

import org.eclipse.microprofile.jwt.JsonWebToken;

@Path("/Sample")
public class SampleEndPoint {

    @GET
    @Path("/helloworld")
    public String helloworld(@Context SecurityContext securityContext) {
        Principal principal = securityContext.getUserPrincipal();
        String caller = principal == null ? "anonymous" : principal.getName();

        return "Hello " + caller;
    }

    @Inject
  JsonWebToken jwt;

  @GET()
  @Path("/subscription")
```

```
@RolesAllowed({"Subscriber"})
public String helloRolesAllowed(@Context SecurityContext ctx) {
    Principal caller =  ctx.getUserPrincipal();
    String name = caller == null ? "anonymous" : caller.getName();
    boolean hasJWT = jwt.getClaimNames() != null;
    String helloReply = String.format("hello + %s, hasJWT: %s", name, hasJWT);

    return helloReply;
}

@Inject
@Claim(standard = Claims.birthdate)
Optional<String> birthdate;

@GET()
@Path("/birthday")
@RolesAllowed({ "Subscriber" })
public String birthday() {
    if (birthdate.isPresent()) {
        LocalDate birthdate = LocalDate.parse(this.birthdate.get().toString());
        LocalDate today = LocalDate.now();
        LocalDate next = birthdate.withYear(today.getYear());
        if (today.equals(next)) {
            return "Happy Birthday";
        }
        if (next.isBefore(today)) {
            next = next.withYear(next.getYear() + 1);
        }

        Period wait = today.until(next);

        return String.format("%d months and %d days until your next birthday.",
wait.getMonths(), wait.getDays());
    }

    return "Sorry, we don't know your birthdate.";

}

}
```

The methods annotated with **@Path** are the JAX-RS endpoints.

The annotation **@Claim** defines a JWT claim.

4. Create a class file **App.java** to enable JAX-RS:

```
package com.example.mpjwt;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

import org.eclipse.microprofile.auth.LoginConfig;
```

```
@ApplicationPath("/rest")
@LoginConfig(authMethod="MP-JWT", realmName="MP JWT Realm")
public class App extends Application {}
```

The annotation **@LoginConfig(authMethod="MP-JWT", realmName="MP JWT Realm")** enables JWT RBAC during deployment.

5. Compile the application with the following Maven command:

```
$ mvn package
```

6. Generate JWT token using the token generator utility:

```
$ mvn exec:java -Dexec.mainClass=org.wildfly.quickstarts.mpjwt.TokenUtil -
Dexec.classpathScope=test -Dexec.args="testUser 2017-09-15 Echoer Subscriber"
```

7. Build and deploy the application using the following Maven command:

```
$ mvn package wildfly:deploy
```

8. Test the application.

   - Call the **Sample/subscription** endpoint using the bearer token:

```
$ curl -H "Authorization: Bearer ey..rg" http://localhost:8080/microprofile-
jwt/rest/Sample/subscription
```

   - Call the **Sample/birthday** endpoint:

```
$ curl -H "Authorization: Bearer ey..rg" http://localhost:8080/microprofile-
jwt/rest/Sample/birthday
```

4.6. ECLIPSE MICROPROFILE METRICS DEVELOPMENT

4.6.1. Creating an Eclipse MicroProfile Metrics application

Create an application that returns the number of requests made to the application.

Procedure

1. Create a class file **HelloService.java** with the following content:

```
package com.example.microprofile.metrics;

public class HelloService {
    String createHelloMessage(String name){
        return "Hello" + name;
    }
}
```

2. Create a class file **HelloWorld.java** with the following content:

```
package com.example.microprofile.metrics;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import org.eclipse.microprofile.metrics.annotation.Counted;

@Path("/")
public class HelloWorld {
@Inject
    HelloService helloService;

@GET
@Path("/json")
    @Produces({ "application/json" })
    @Counted(name = "requestCount",
     absolute = true,
description = "Number of times the getHelloWorldJSON was requested")
    public String getHelloWorldJSON() {
        return "{\"result\":\"" + helloService.createHelloMessage("World") + "\"}";
    }
}
```

3. Update the **pom.xml** file to include the following dependency:

```
<dependency>
    <groupId>org.eclipse.microprofile.metrics</groupId>
    <artifactId>microprofile-metrics-api</artifactId>
    <scope>provided</scope>
</dependency>
```

4. Build the application using the following Maven command:

```
$ mvn clean install wildfly:deploy
```

5. Test the metrics:

   a. Issue the following command in the CLI:

   ```
   $ curl -v http://localhost:9990/metrics |  grep request_count | grep helloworld-rs-metrics
   ```

   Expected output:

   ```
   jboss_undertow_request_count_total{deployment="helloworld-rs-
   metrics.war",servlet="org.jboss.as.quickstarts.rshelloworld.JAXActivator",subdeployment="h
   elloworld-rs-metrics.war",microprofile_scope="vendor"} 0.0
   ```

   b. In a browser, navigate to the URL http://localhost:8080/helloworld-rs/rest/json.

   c. Re-Issue the following command in the CLI:

   ```
   $ curl -v http://localhost:9990/metrics |  grep request_count | grep helloworld-rs-metrics
   ```

Expected output:

```
jboss_undertow_request_count_total{deployment="helloworld-rs-metrics.war",servlet="org.jboss.as.quickstarts.rshelloworld.JAXActivator",subdeployment="helloworld-rs-metrics.war",microprofile_scope="vendor"} 1.0
```

4.7. DEVELOPING AN ECLIPSE MICROPROFILE OPENAPI APPLICATION

4.7.1. Enabling Eclipse MicroProfile OpenAPI

The **microprofile-openapi-smallrye** subsystem is provided in the **standalone-microprofile.xml** configuration. However, JBoss EAP XP uses the **standalone.xml** by default. You must include the subsystem in **standalone.xml** to use it.

Alternatively, you can follow the procedure Updating standalone configurations with Eclipse MicroProfile subsystems and extensions to update the **standalone.xml** configuration file.

Procedure

1. Enable the MicroProfile OpenAPI smallrye extension in JBoss EAP:

```
/extension=org.wildfly.extension.microprofile.openapi-smallrye:add()
```

2. Enable the **microprofile-openapi-smallrye** subsystem using the following management command:

```
/subsystem=microprofile-openapi-smallrye:add()
```

3. Reload the server.

```
reload
```

The **microprofile-openapi-smallrye** subsystem is enabled.

4.7.2. Configuring Maven project for Eclipse MicroProfile OpenAPI

Create a Maven project to set up the dependencies for creating an Eclipse MicroProfile OpenAPI application.

Prerequisites

- Maven is installed.
- JBoss EAP Maven repository is configured.

Procedure

1. Initialize the project:

```
mvn archetype:generate \
    -DgroupId=com.example.microprofile.openapi \
```

```
    -DartifactId=microprofile-openapi\
    -DarchetypeGroupId=org.apache.maven.archetypes \
    -DarchetypeArtifactId=maven-archetype-webapp \
    -DinteractiveMode=false
cd microprofile-openapi
```

The command creates the directory structure for the project and the **pom.xml** configuration file.

2. Edit the **pom.xml** configuration file to contain:

```
<?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.microprofile.openapi</groupId>
  <artifactId>microprofile-openapi</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>microprofile-openapi Maven Webapp</name>
  <!-- Update the value with the URL of the project -->
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <version.server.bom>2.0.0.GA</version.server.bom>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.bom</groupId>
        <artifactId>jboss-eap-xp-microprofile</artifactId>
        <version>${version.server.bom}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>org.jboss.spec.javax.ws.rs</groupId>
      <artifactId>jboss-jaxrs-api_2.1_spec</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
```

```
        <!-- Set the name of the archive -->
        <finalName>${project.artifactId}</finalName>
        <plugins>
            <plugin>
                <artifactId>maven-clean-plugin</artifactId>
                <version>3.1.0</version>
            </plugin>
            <!-- see http://maven.apache.org/ref/current/maven-core/default-
bindings.html#Plugin_bindings_for_war_packaging -->
            <plugin>
                <artifactId>maven-resources-plugin</artifactId>
                <version>3.0.2</version>
            </plugin>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.0</version>
            </plugin>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.1</version>
            </plugin>
            <plugin>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.2.2</version>
            </plugin>
            <plugin>
                <artifactId>maven-install-plugin</artifactId>
                <version>2.5.2</version>
            </plugin>
            <plugin>
                <artifactId>maven-deploy-plugin</artifactId>
                <version>2.8.2</version>
            </plugin>
            <!-- Allows to use mvn wildfly:deploy -->
            <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Use the **pom.xml** configuration file and directory structure to create an application.

Additional resources

- For information about configuring the JBoss EAP Maven repository, see Configuring the JBoss EAP Maven repository with the POM file.

4.7.3. Creating an Eclipse MicroProfile OpenAPI application

Create an application that returns an OpenAPI v3 document.

Prerequisites

- Maven project is configured for creating an Eclipse MicroProfile OpenAPI application.

Procedure

1. Create the directory to store class files:

```
$ mkdir -p APPLICATION_ROOT/src/main/java/com/example/microprofile/openapi/
```

   APPLICATION_ROOT is the directory containing the **pom.xml** configuration file for the application.

2. Navigate to the new directory:

```
$ cd APPLICATION_ROOT/src/main/java/com/example/microprofile/openapi/
```

   All the class files in the following steps must be created in this directory.

3. Create the class file **InventoryApplication.java** with the following content:

```
package com.example.microprofile.openapi;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/inventory")
public class InventoryApplication extends Application {
}
```

   This class serves as the REST endpoint for the application.

4. Create a class file **Fruit.java** with the following content:

```
package com.example.microprofile.openapi;

public class Fruit {

    private final String name;
    private final String description;

    public Fruit(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public String getName() {
        return this.name;
    }

    public String getDescription() {
        return this.description;
    }
}
```

5. Create a class file **FruitResource.java** with the following content:

```
package com.example.microprofile.openapi;

import java.util.Collections;
import java.util.LinkedHashMap;
import java.util.Set;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/fruit")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class FruitResource {

    private final Set<Fruit> fruits =
Collections.newSetFromMap(Collections.synchronizedMap(new LinkedHashMap<>()));

    public FruitResource() {
        this.fruits.add(new Fruit("Apple", "Winter fruit"));
        this.fruits.add(new Fruit("Pineapple", "Tropical fruit"));
    }

    @GET
    public Set<Fruit> all() {
        return this.fruits;
    }

    @POST
    public Set<Fruit> add(Fruit fruit) {
        this.fruits.add(fruit);
        return this.fruits;
    }

    @DELETE
    public Set<Fruit> remove(Fruit fruit) {
        this.fruits.removeIf(existingFruit ->
existingFruit.getName().contentEquals(fruit.getName()));
        return this.fruits;
    }
}
```

6. Navigate to the root directory of the application:

```
$ cd APPLICATION_ROOT
```

7. Build and deploy the application using the following Maven command:

```
$ mvn wildfly:deploy
```

8. Test the application.

- Access the OpenAPI documentation of the sample application using **curl**:

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

- The following output is returned:

```
openapi: 3.0.1
info:
  title: Archetype Created Web Application
  version: "1.0"
servers:
- url: /microprofile-openapi
paths:
  /inventory/fruit:
    get:
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Fruit'
    post:
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Fruit'
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Fruit'
    delete:
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Fruit'
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Fruit'
components:
  schemas:
```

```
Fruit:
  type: object
  properties:
    description:
      type: string
    name:
      type: string
```

Additional Resources

- For a list of annotations defined in MicroProfile SmallRye OpenAPI, see MicroProfile OpenAPI annotations.

4.7.4. Configuring JBoss EAP to serve a static OpenAPI document

Configure JBoss EAP to serve a static OpenAPI document that describes the REST services for the host.

When JBoss EAP is configured to serve a static OpenAPI document, the static OpenAPI document is processed before any JAX-RS and MicroProfile OpenAPI annotations.

In a production environment, disable annotation processing when serving a static document. Disabling annotation processing ensures that an immutable and versioned API contract is available for clients.

Procedure

1. Create a directory in the application source tree:

   ```
   $ mkdir APPLICATION_ROOT/src/main/webapp/META-INF
   ```

   APPLICATION_ROOT is the directory containing the **pom.xml** configuration file for the application.

2. Query the OpenAPI endpoint, redirecting the output to a file:

   ```
   $ curl http://localhost:8080/openapi?format=JSON > src/main/webapp/META-INF/openapi.json
   ```

   By default, the endpoint serves a YAML document, **format=JSON** specifies that a JSON document is returned.

3. Configure the application to skip annotation scanning when processing the OpenAPI document model:

   ```
   $ echo "mp.openapi.scan.disable=true" > APPLICATION_ROOT/src/main/webapp/META-INF/microprofile-config.properties
   ```

4. Rebuild the application:

   ```
   $ mvn clean install
   ```

5. Deploy the application again using the following management CLI commands:

   a. Undeploy the application:

```
undeploy microprofile-openapi.war
```

b. Deploy the application:

```
deploy APPLICATION_ROOT/target/microprofile-openapi.war
```

JBoss EAP now serves a static OpenAPI document at the OpenAPI endpoint.

4.8. ECLIPSE MICROPROFILE REST CLIENT DEVELOPMENT

4.8.1. A comparison between MicroProfile REST client and JAX-RS syntaxes

The MicroProfile REST client enables a version of distributed object communication, which is also implemented in CORBA, Java Remote Method Invocation (RMI), the JBoss Remoting Project, and RESTEasy. For example, consider the resource:

```
@Path("resource")
public class TestResource {
   @Path("test")
   @GET
   String test() {
      return "test";
   }
 }
```

The following example demonstrates using the JAX-RS native way to access the **TestResource** class:

```
Client client = ClientBuilder.newClient();
String response = client.target("http://localhost:8081/test").request().get(String.class);
```

However, Microprofile REST client supports a more intuitive syntax by directly calling the **test()** method as the following example demonstrates:

```
@Path("resource")
public interface TestResourceIntf {
    @Path("test")
    @GET
    public String test();
}

TestResourceIntf service = RestClientBuilder.newBuilder()
                              .baseUrl(http://localhost:8081/))
                              .build(TestResourceIntf.class);
String s = service.test();
```

In the preceding example, making calls on the **TestResource** class becomes much easier with the **TestResourceIntf** class, as illustrated by the call **service.test()**.

The following example is a more elaborate version of the **TestResourceIntf** class:

```
@Path("resource")
public interface TestResourceIntf2 {
```

```
  @Path("test/{path}")mes("text/plain")
  @Produces("text/html")
  @POST
  public String test(@PathParam("path") String path, @QueryParam("query") String query, String
entity);
}
```

Calling the **service.test("p", "q", "e")** method results in an HTTP message as shown in the following example:

```
POST /resource/test/p/?query=q HTTP/1.1
Accept: text/html
Content-Type: text/plain
Content-Length: 1

e
```

4.8.2. Programmatic registration of providers in MicroProfile REST client

With the MicroProfile REST client, you can configure the client environment by registering providers. For example:

```
TestResourceIntf service = RestClientBuilder.newBuilder()
                    .baseUrl(http://localhost:8081/))
                    .register(MyClientResponseFilter.class)
                    .register(MyMessageBodyReader.class)
                    .build(TestResourceIntf.class);
```

4.8.3. Declarative registration of providers in MicroProfile REST client

Use the MicroProfile REST client to register providers declaratively by adding the **org.eclipse.microprofile.rest.client.annotation.RegisterProvider** annotation to the target interface, as shown in the following example:

```
@Path("resource")
@RegisterProvider(MyClientResponseFilter.class)
@RegisterProvider(MyMessageBodyReader.class)
public interface TestResourceIntf2 {
  @Path("test/{path}")
  @Consumes("text/plain")
  @Produces("text/html")
  @POST
  public String test(@PathParam("path") String path, @QueryParam("query") String query, String
entity);
}
```

Declaring the **MyClientResponseFilter** class and the **MyMessageBodyReader** class with annotations eliminates the need to call the **RestClientBuilder.register()** method.

4.8.4. Declarative specification of headers in MicroProfile REST client

You can specify a header for an HTTP request in the following ways:

- By annotating one of the resource method parameters.
- By declaratively using the **org.eclipse.microprofile.rest.client.annotation.ClientHeaderParam** annotation.

The following example illustrates setting a header by annotating one of the resource method parameters with the annotation **@HeaderValue**:

```
@POST
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
String contentLang(@HeaderParam(HttpHeaders.CONTENT_LANGUAGE) String contentLanguage,
String subject);
```

The following example illustrates setting a header using the **org.eclipse.microprofile.rest.client.annotation.ClientHeaderParam** annotation:

```
@POST
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
@ClientHeaderParam(name=HttpHeaders.CONTENT_LANGUAGE, value="{getLanguage}")
String contentLang(String subject);

default String getLanguage() {
  return ...;
}
```

4.8.5. Propagation of headers on the server in MicroProfile REST client

An instance of **org.eclipse.microprofile.rest.client.ext.ClientHeadersFactory**, if activated, can do a bulk transfer of incoming headers to an outgoing request. The default instance **org.eclipse.microprofile.rest.client.ext.DefaultClientHeadersFactoryImpl** returns a map consisting of those incoming headers that are listed in the comma-separated configuration property **org.eclipse.microprofile.rest.client.propagateHeaders**.

The following are the rules for instantiating the **ClientHeadersFactory** interface:

- A **ClientHeadersFactory** instance invoked in the context of a JAX-RS request can support injection of fields and methods annotated with **@Context**.
- A **ClientHeadersFactory** instance that is managed by CDI must use the appropriate CDI-managed instance. It must also support the **@Inject** injection.

The **org.eclipse.microprofile.rest.client.ext.ClientHeadersFactory** interface is defined as follows:

```
public interface ClientHeadersFactory {

/**
 * Updates the HTTP headers to send to the remote service. Note that providers
 * on the outbound processing chain could further update the headers.
 *
 * @param incomingHeaders - the map of headers from the inbound JAX-RS request. This will
 * be an empty map if the associated client interface is not part of a JAX-RS request.
 * @param clientOutgoingHeaders - the read-only map of header parameters specified on the
 * client interface.
```

```
 * @return a map of HTTP headers to merge with the clientOutgoingHeaders to be sent to
 * the remote service.
 */
MultivaluedMap<String, String> update(MultivaluedMap<String, String> incomingHeaders,
                                       MultivaluedMap<String, String> clientOutgoingHeaders);
}
```

Additional resources

- ClientHeadersFactory Javadoc

4.8.6. ResponseExceptionMapper in MicroProfile REST client

The **org.eclipse.microprofile.rest.client.ext.ResponseExceptionMapper** class is the client-side inverse of the **javax.ws.rs.ext.ExceptionMapper** class, which is defined in JAX-RS. The **ExceptionMapper.toResponse()** method turns an **Exception** class thrown during the server-side processing into a **Response** class. The **ResponseExceptionMapper.toThrowable()** method turns a **Response** class received on the client-side with an HTTP error status into an **Exception** class.

You can register the **ResponseExceptionMapper** class either programmatically or declaratively. In the absence of a registered **ResponseExceptionMapper** class, a default **ResponseExceptionMapper** class maps any response with status **>= 400** to a **WebApplicationException** class.

4.8.7. Context dependency injection with MicroProfile REST client

In MicroProfile REST client, you must annotate any interface that is managed as a CDI bean with the **@RegisterRestClient** class. For example:

```
@Path("resource")
@RegisterProvider(MyClientResponseFilter.class)
public static class TestResourceImpl {
    @Inject TestDataBase db;

    @Path("test/{path}")
    @Consumes("text/plain")
    @Produces("text/html")
    @POST
    public String test(@PathParam("path") String path, @QueryParam("query")
    String query, String entity) {
      return db.getByName(query);
    }
  }
  @Path("database")
  @RegisterRestClient
  public interface TestDataBase {

    @Path("")
    @POST
    public String getByName(String name);
  }
```

Here, the MicroProfile REST client implementation creates a client for a **TestDataBase** class service, allowing easy access by the **TestResourceImpl** class. However, it does not include the information about the path to the **TestDataBase** class implementation. This information can be supplied by the optional **@RegisterProvider** parameter **baseUri**:

```
@Path("database")
@RegisterRestClient(baseUri="https://localhost:8080/webapp")
public interface TestDataBase {
  @Path("")
  @POST
  public String getByName(String name);
}
```

This indicates that you can access the implementation of **TestDataBase** at https://localhost:8080/webapp. You can also supply the information externally with the following system variable:

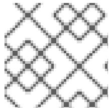
```
<fully qualified name of TestDataBase>/mp-rest/url=<URL>
```

For example, the following command indicates that you can access an implementation of the **com.bluemonkeydiamond.TestDatabase** class at https://localhost:8080/webapp:

```
com.bluemonkeydiamond.TestDatabase/mp-rest/url=https://localhost:8080/webapp
```

CHAPTER 5. BUILD AND RUN MICROSERVICES APPLICATIONS ON THE OPENSHIFT IMAGE FOR JBOSS EAP XP

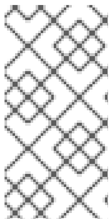
You can build and run your microservices applications on the OpenShift image for JBoss EAP XP.



NOTE

JBoss EAP XP is supported only on OpenShift 4 and later versions.

Use the following workflow to build and run a microservices application on the OpenShift image for JBoss EAP XP by using the source-to-image (S2I) process.



NOTE

The OpenShift images for JBoss EAP XP 2.0.0 provide a default standalone configuration file, which is based on the **standalone-microprofile-ha.xml** file. For more information about the server configuration files included in JBoss EAP XP, see the *Standalone server configuration files* section.

This workflow uses the **microprofile-config** quickstart as an example. The quickstart provides a small, specific working example that can be used as a reference for your own project. See the **microprofile-config** quickstart that ships with JBoss EAP XP 2.0.0 for more information.

Additional resources

- For more information about the server configuration files included in JBoss EAP XP, see Standalone server configuration files.

5.1. PREPARING OPENSHIFT FOR APPLICATION DEPLOYMENT

Prepare OpenShift for application deployment.

Prerequisites

You have installed an operational OpenShift instance. For more information, see the *Installing and Configuring OpenShift Container Platform Clusters* book on Red Hat Customer Portal.

Procedure

1. Log in to your OpenShift instance using the **oc login** command.
2. Create a new project in OpenShift.
   A project allows a group of users to organize and manage content separately from other groups. You can create a project in OpenShift using the following command.

```
$ oc new-project PROJECT_NAME
```

   For example, for the **microprofile-config** quickstart, create a new project named **eap-demo** using the following command.

```
$ oc new-project eap-demo
```

–

5.2. CONFIGURING AUTHENTICATION TO THE RED HAT CONTAINER REGISTRY

Before you can import and use the OpenShift image for JBoss EAP XP, you must configure authentication to the Red Hat Container Registry.

Create an authentication token using a registry service account to configure access to the Red Hat Container Registry. You need not use or store your Red Hat account's username and password in your OpenShift configuration when you use an authentication token.

Procedure

1. Follow the instructions on Red Hat Customer Portal to create an authentication token using a Registry Service Account management application.
2. Download the YAML file containing the OpenShift secret for the token.
   You can download the YAML file from the **OpenShift Secret** tab on your token's **Token Information** page.
3. Create the authentication token secret for your OpenShift project using the YAML file that you downloaded:

   ```
   oc create -f 1234567_myserviceaccount-secret.yaml
   ```

4. Configure the secret for your OpenShift project using the following commands, replacing the secret name below with the name of your secret created in the previous step.

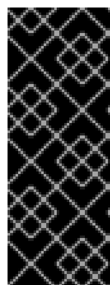
   ```
   oc secrets link default 1234567-myserviceaccount-pull-secret --for=pull
   oc secrets link builder 1234567-myserviceaccount-pull-secret --for=pull
   ```

Additional resources

- Configuring authentication to the Red Hat Container Registry
- Registry Service Account management application
- Configuring access to secured registries

5.3. IMPORTING THE LATEST OPENSHIFT IMAGESTREAMS AND TEMPLATES FOR JBOSS EAP XP

Import the latest OpenShift imagestreams and templates for JBoss EAP XP.



IMPORTANT

OpenJDK 8 images and imagestreams on OpenShift are deprecated.

The images and imagestreams are still supported on OpenShift. However, no enhancements are made to these images and imagestreams and they might be removed in the future. Red Hat continues to provide full support and bug fixes OpenJDK 8 images and imagestreams under its standard support terms and conditions.

Procedure

1. Use one of the following commands to import the latest JDK 8 and JDK 11 imagestreams and templates for the OpenShift image for JBoss EAP XP into your OpenShift project's namespace.

   a. Import JDK 8 imagestreams:

   ```
   oc replace --force -f https://raw.githubusercontent.com/jboss-container-images/jboss-eap-openshift-templates/eap-xp2/jboss-eap-xp2-openjdk8-openshift.json
   ```

   This command imports the following imagestreams and templates:

   - The JDK 8 builder imagestream: jboss-eap-xp2-openjdk8-openshift
   - The JDK 8 runtime imagestream: jboss-eap-xp2-openjdk8-runtime-openshift

   b. Import JDK 11 imagestream:

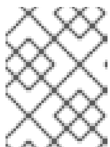
   ```
   oc replace --force -f https://raw.githubusercontent.com/jboss-container-images/jboss-eap-openshift-templates/eap-xp2/jboss-eap-xp2-openjdk11-openshift.json
   ```

   This command imports the following imagestreams and templates:

   - The JDK 11 builder imagestream: jboss-eap-xp2-openjdk11-openshift
   - The JDK 11 runtime imagestream: jboss-eap-xp2-openjdk11-runtime-openshift

   c. Import the JDK 8 and JDK 11 templates:

   ```
   for resource in \
   eap-xp2-basic-s2i.json \
   eap-xp2-third-party-db-s2i.json
   do
   oc replace --force -f https://raw.githubusercontent.com/jboss-container-images/jboss-eap-openshift-templates/eap-xp2/templates/${resource}
   done
   ```

   NOTE

   The JBoss EAP XP imagestreams and templates imported using the above command are only available within that OpenShift project.

2. If you have administrative access to the general **openshift** namespace and want the imagestreams and templates to be accessible by all projects, add **-n openshift** to the **oc replace** line of the command. For example:

   ```
   ...
   oc replace -n openshift --force -f \
   ...
   ```

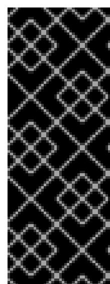
3. If you want to import the imagestreams and templates into a different project, add the **-n *PROJECT_NAME*** to the **oc replace** line of the command. For example:

```
...
oc replace -n PROJECT_NAME --force -f
...
```

If you use the cluster-samples-operator, see the OpenShift documentation on configuring the cluster samples operator. See https://docs.openshift.com/container-platform/latest/openshift_images/configuring-samples-operator.html for details about configuring the cluster samples operator.

5.4. DEPLOYING A JBOSS EAP XP SOURCE-TO-IMAGE (S2I) APPLICATION ON OPENSHIFT

Deploy a JBoss EAP XP source-to-image (S2I) application on OpenShift.



IMPORTANT

OpenJDK 8 images and imagestreams on OpenShift are deprecated.

The images and imagestreams are still supported on OpenShift. However, no enhancements are made to these images and imagestreams and they might be removed in the future. Red Hat continues to provide full support and bug fixes for OpenJDK 8 images and imagestreams under its standard support terms and conditions.

Prerequisites

- Optional: A template can specify default values for many template parameters, and you might have to override some, or all, of the defaults. To see template information, including a list of parameters and any default values, use the command **oc describe template TEMPLATE_NAME**.

Procedure

1. Create a new OpenShift application using the JBoss EAP XP image and your Java application's source code. Use one of the provided JBoss EAP XP templates for S2I builds.

```
$ oc new-app --template=eap-xp2-basic-s2i \ (1)
 -p EAP_IMAGE_NAME=jboss-eap-xp2-openjdk8-openshift:latest \
 -p EAP_RUNTIME_IMAGE_NAME=jboss-eap-xp2-openjdk8-runtime-openshift:latest \
 -p IMAGE_STREAM_NAMESPACE=eap-demo \ (2)
 -p SOURCE_REPOSITORY_URL=https://github.com/jboss-developer/jboss-eap-quickstarts
\ (3)
 -p SOURCE_REPOSITORY_REF=xp-2.0.x \ (4)
 -p CONTEXT_DIR=microprofile-config (5)
```

(1) The template to use. The application image is tagged with the **latest** tag.

(2) The latest images streams and templates were imported into the project's namespace, so you must specify the namespace of where to find the imagestream. This is usually the project's name.

(3) URL to the repository containing the application source code.

(4)

The Git repository reference to use for the source code. This can be a Git branch or tag reference.

5 The directory within the source repository to build.

As another example, to deploy the **microprofile-config** quickstart using the JDK 11 runtime image enter the following command. The command uses the **eap-xp2-basic-s2i** template in the **eap-demo** project, created in the Preparing OpenShift for application deployment section, with the **microprofile-config** source code on GitHub.

```
$ oc new-app --template=eap-xp2-basic-s2i \ 1
 -p EAP_IMAGE_NAME=jboss-eap-xp2-openjdk11-openshift:latest \
 -p EAP_RUNTIME_IMAGE_NAME=jboss-eap-xp2-openjdk11-runtime-openshift:latest \
 -p IMAGE_STREAM_NAMESPACE=eap-demo \ 2
 -p SOURCE_REPOSITORY_URL=https://github.com/jboss-developer/jboss-eap-quickstarts
\ 3
 -p SOURCE_REPOSITORY_REF=xp-2.0.x \ 4
 -p CONTEXT_DIR=microprofile-config 5
```

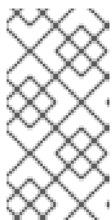
1 The template to use. The application image is tagged with the **latest** tag.

2 The latest imagestreams and templates were imported into the project's namespace, so you must specify the namespace where to find the imagestream. This is usually the project's name.

3 URL to the repository containing the application source code.

4 The Git repository reference to use for the source code. This can be a Git branch or tag reference.

5 The directory within the source repository to build.



NOTE

A template can specify default values for many template parameters, and you might have to override some, or all, of the defaults. To see template information, including a list of parameters and any default values, use the command **oc describe template *TEMPLATE_NAME***.

You might also want to configure environment variables when creating your new OpenShift application.

2. Retrieve the name of the build configurations.

```
$ oc get bc -o name
```

3. Use the name of the build configurations from the previous step to view the Maven progress of the builds.

```
$ oc logs -f buildconfig/${APPLICATION_NAME}-build-artifacts

…
Push successful
```

```
$ oc logs -f buildconfig/${APPLICATION_NAME}
...
Push successful
```

For example, for the **microprofile-config**, the following command shows the progress of the Maven builds.

```
$ oc logs -f buildconfig/eap-xp2-basic-app-build-artifacts

...
Push successful
$ oc logs -f buildconfig/eap-xp2-basic-app
...
Push successful
```

Additional resources

- Importing the latest OpenShift imagestreams and templates for JBoss EAP XP
- Preparing OpenShift for application deployment

5.5. COMPLETING POST-DEPLOYMENT TASKS FOR JBOSS EAP XP SOURCE-TO-IMAGE (S2I) APPLICATION

Depending on your application, you might need to complete some tasks after your OpenShift application has been built and deployed.

Examples of post-deployment tasks include the following:

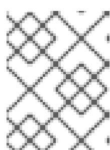
- Exposing a service so that the application is viewable from outside of OpenShift.
- Scaling your application to a specific number of replicas.

Procedure

1. Get the service name of your application using the following command.

```
$ oc get service
```

2. **Optional**: Expose the main service as a route so you can access your application from outside of OpenShift. For example, for the **microprofile-config** quickstart, use the following command to expose the required service and port.



NOTE

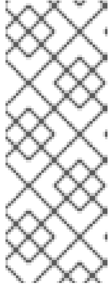
If you used a template to create the application, the route might already exist. If it does, continue on to the next step.

```
$ oc expose service/eap-xp2-basic-app --port=8080
```

3. Get the URL of the route.

```
$ oc get route
```

4. Access the application in your web browser using the URL. The URL is the value of the **HOST/PORT** field from previous command's output.



NOTE

For JBoss EAP XP 2.0.0 GA distribution, the Microprofile Config quickstart does not reply to HTTPS GET requests to the application's root context. This enhancement is only available in the {JBossXPShortName101} GA distribution.

For example, to interact with the Microprofile Config application, the URL might be **http://*HOST_PORT_Value*/config/value** in your browser.

If your application does not use the JBoss EAP root context, append the context of the application to the URL. For example, for the **microprofile-config** quickstart, the URL might be **http://*HOST_PORT_VALUE*/microprofile-config/**.

5. Optionally, you can scale up the application instance by running the following command. This command increases the number of replicas to 3.

```
$ oc scale deploymentconfig DEPLOYMENTCONFIG_NAME --replicas=3
```

For example, for the **microprofile-config** quickstart, use the following command to scale up the application.

```
$ oc scale deploymentconfig/eap-xp2-basic-app --replicas=3
```

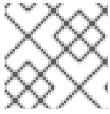
Additional Resources

For more information about JBoss EAP XP Quickstarts, see the Use the Quickstarts section in the *Using Eclipse MicroProfile in JBoss EAP* guide.

CHAPTER 6. CAPABILITY TRIMMING

When building an image that includes JBoss EAP, you can control the JBoss EAP features and subsystems to include in the image.

The default JBoss EAP server included in S2I images includes the complete server and all features. You might want to trim the capabilities included in the provisioned server. For example, you might want to reduce the security exposure of the provisioned server, or you might want to reduce the memory footprint so it is more appropriate for a microservice container.



NOTE

Capability trimming is supported only on OpenShift or when building a bootable JAR.

6.1. PROVISION A CUSTOM JBOSS EAP SERVER

To provision a custom server with trimmed capabilities, pass the **GALLEON_PROVISION_LAYERS** environment variable during the S2I build phase.

The value of the environment variable is a comma-separated list of the layers to provision to build the server.

For example, if you specify the environment variable as **GALLEON_PROVISION_LAYERS=jaxrs-server,sso**, a JBoss EAP server is provisioned with the following capabilities:

- A servlet container
- The ability to configure a datasource
- The **jaxrs**, **weld**, and **jpa** subsystems
- Red Hat SSO integration

6.1.1. Available JBoss EAP layers

Red Hat makes available a number of layers to customize provisioning of the JBoss EAP server in OpenShift or a bootable JAR.

Three layers are base layers that provide core functionality. The other layers are decorator layers that enhance the base layers with additional capabilities.

Most decorator layers can be used to build both S2I images in OpenShift and bootable JAR. A few layers do not support S2I images; the description of the layer notes this limitation.

6.1.2. Base layers

Each base layer includes core functionality for a typical server user case.

datasources-web-server
This layer includes a servlet container and the ability to configure a datasource.

This layer does not include MicroProfile capabilities.

The following Jakarta EE specifications are supported in this layer:

- Jakarta JSON Processing 1.1
- Jakarta JSON Binding 1.0
- Jakarta Servlet 4.0
- Jakarta Expression Language 3.0
- Jakarta Server Pages 2.3
- Jakarta Standard Tag Library 1.2
- Jakarta Concurrency 1.1
- Jakarta Annotations 1.3
- Jakarta XML Binding 2.3
- Jakarta Debugging Support for Other Languages 1.0
- Jakarta Transaction 1.3
- Jakarta Connector API 1.7

jaxrs-server
This layer enhances the **datasources-web-server** layer with the following JBoss EAP subsystems:

- **jaxrs**
- **weld**
- **jpa**

This layer also adds Infinispan-based second-level entity caching locally in the container.

The following MicroProfile capability is included in this layer:

- MicroProfile REST Client

The following Jakarta EE specifications are supported in this layer in addition to those supported in the **datasources-web-server** layer:

- Jakarta Contexts and Dependency Injection 2.0
- Jakarta Bean Validation 2.0
- Jakarta Interceptors 1.2
- Jakarta RESTful Web Services 2.1
- Jakarta Persistence 2.2

cloud-server
This layer enhances the **jaxrs-server** layer with the following JBoss EAP subsystems:

- **resource-adapters**
- **messaging-activemq** (remote broker messaging, not embedded messaging)

This layer also adds the following observability features to the **jaxrs-server** layer:

- MicroProfile Health
- MicroProfile Metrics
- MicroProfile Config
- MicroProfile OpenTracing

The following Jakarta EE specification is supported in this layer in addition to those supported in the **jaxrs-server** layer:

- Jakarta Security 1.0

6.1.3. Decorator layers

Decorator layers are not used alone. You can configure one or more decorator layers with a base layer to deliver additional functionality.

ejb-lite
This decorator layer adds a minimal EJB/Jakarta Enterprise Bean implementation to the provisioned server. The following support is not included in this layer:

- IIOP integration
- MDB instance pool
- Remote connector resource

This layer is only supported when building a bootable JAR. This layer is not supported when using S2I.

ejb
This decorator layer extends the **ejb-lite** layer. This layer adds the following support to the provisioned server, in addition to the base functionality included in the **ejb-lite** layer:

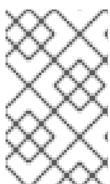
- MDB instance pool
- Remote connector resource

Use this layer if you want to use message-driven beans (MDBs) or EJB remoting capabilities, or both. If you do not need these capabilities, use the **ejb-lite** layer.

This layer is only supported when building a bootable JAR. This layer is not supported when using S2I.

ejb-local-cache
This decorator layer adds local caching support for EJBs/Jakarta Enterprise Beans to the provisioned server.

Dependencies: You can only include this layer if you have included the **ejb-lite** layer or the **ejb** layer.



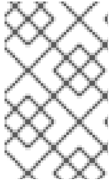
NOTE

This layer is not compatible with the **ejb-dist-cache** layer. If you include the **ejb-dist-cache** layer, you cannot include the **ejb-local-cache** layer. If you include both layers, the resulting build may include an unexpected EJB configuration.

This layer is only supported when building a bootable JAR. This layer is not supported when using S2I.

ejb-dist-cache
This decorator layer adds distributed caching support for EJBs/Jakarta Enterprise Beans to the provisioned server.

Dependencies: You can only include this layer if you have included the **ejb-lite** layer or the **ejb** layer.



NOTE

This layer is not compatible with the **ejb-local-cache** layer. If you include the **ejb-dist-cache** layer, you cannot include the **ejb-local-cache** layer. If you include both layers, the resulting build may result in an unexpected configuration.

This layer is only supported when building a bootable JAR. This layer is not supported when using S2I.

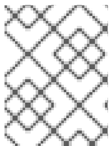
jdr
This decorator layer adds the JBoss Diagnostic Reporting (**jdr**) subsystem to gather diagnostic data when requesting support from Red Hat.

This layer is only supported when building a bootable JAR. This layer is not supported when using S2I.

jpa
This decorator layer adds persistence capabilities for a single-node server. Note that distributed caching only works if the servers are able to form a cluster.

The layer adds Hibernate libraries to the provisioned server, with the following support:

- Configurations of the **jpa** subsystem
- Configurations of the **infinispan** subsystem
- A local Hibernate cache container



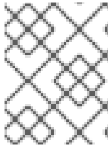
NOTE

This layer is not compatible with the **jpa-distributed** layer. If you include the **jpa** layer, you cannot include the **jpa-distributed** layer.

This layer is only supported when building a bootable JAR. This layer is not supported when using S2I.

jpa-distributed
This decorator layer adds persistence capabilities for servers operating in a cluster. The layer adds Hibernate libraries to the provisioned server, with the following support:

- Configurations of the **jpa** subsystem
- Configurations of the **infinispan** subsystem
- A local Hibernate cache container
- Invalidation and replication Hibernate cache containers
- Configuration of the **jgroups** subsystem



NOTE

This layer is not compatible with the **jpa** layer. If you include the **jpa** layer, you cannot include the **jpa-distributed** layer.

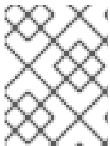
This layer is only supported when building a bootable JAR. This layer is not supported when using S2I.

jsf
This decorator layer adds the **jsf** subsystem to the provisioned server.

This layer is only supported when building a bootable JAR. This layer is not supported when using S2I.

microprofile-platform
This decorator layer adds the following Eclipse MicroProfile capabilities to the provisioned server:

- MicroProfile Config
- MicroProfile Fault Tolerance
- MicroProfile Health
- MicroProfile JWT
- MicroProfile Metrics
- MicroProfile OpenAPI
- MicroProfile OpenTracing

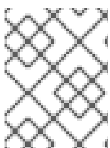


NOTE

This layer includes MicroProfile capabilities that are also included in the **observability** layer. If you include this layer, you do not need to include the **observability** layer.

observability
This decorator layer adds the following observability features to the provisioned server:

- MicroProfile Health
- MicroProfile Metrics
- MicroProfile Config
- MicroProfile OpenTracing



NOTE

This layer is built in to the **cloud-server** layer. You do not need to add this layer to the **cloud-server** layer.

remote-activemq
This decorator layer adds the ability to communicate with a remote ActiveMQ broker to the provisioned server, integrating messaging support.

The pooled connection factory configuration specifies **guest** as the value for the **user** and **password** attributes. You can use a CLI script to change these values at runtime.

This layer is only supported when building a bootable JAR. This layer is not supported when using S2I.

sso
This decorator layer adds Red Hat Single Sign-On integration to the provisioned server.

This layer should only be used when provisioning a server using S2I.

web-console
This decorator layer adds the management console to the provisioned server.

This layer is only supported when building a bootable JAR. This layer is not supported when using S2I.

web-clustering
This layer adds embedded Infinispan-based web session clustering to the provisioned server.

webservices
This layer adds web services functionality to the provisioned server, supporting Jakarta web services deployments.

This layer is only supported when building a bootable JAR. This layer is not supported when using S2I.

Additional resources

- Pooled Connection Factory Attributes

6.2. PROVISIONING USER-DEVELOPED LAYERS IN JBOSS EAP

In addition to provisioning layers available from Red Hat, you can provision custom layers you develop.

NOTE

Use of custom layers is only supported on OpenShift. Custom layers are not supported when building a bootable JAR.

Procedure

1. Build a custom layer using the Galleon Maven plugin.
   For more information, see Building Custom Layers for JBoss EAP .
2. Deploy the custom layer to an accessible Maven repository.
3. Create a custom provisioning file to reference the user-defined layer and supported JBoss EAP layers and store it in your application directory.
   For more information, see Custom Provisioning Files for JBoss EAP .
4. Run the S2I process to provision a JBoss EAP server in OpenShift.
   For more information, see Building an Application Provisioned with User-developed Layers .

6.2.1. Building custom layers for JBoss EAP

Create your custom layer feature pack as a Maven project.

Procedure

1. Custom layers depend on at least a base layer. Select the base layer that provides the capabilities you need for your custom layer.

2. Within the Maven project, create your layer content in the directory **src/main/resources**.
   For example, to create layers to provision support for PostgreSQL and a PostgreSQL datasource, in the directory **src/main/resources** create the **layers/standalone** subdirectories. The **standalone** subdirectory includes the following content.

   - **postgresql-driver**
     This directory contains a **layer-spec.xml** file with the following content:

```
<?xml version="1.0" ?>
<layer-spec xmlns="urn:jboss:galleon:layer-spec:1.0" name="postgresql-driver">
  <feature spec="subsystem.datasources">
    <feature spec="subsystem.datasources.jdbc-driver">
      <param name="driver-name" value="postgresql"/>
      <param name="jdbc-driver" value="postgresql"/>
      <param name="driver-xa-datasource-class-name"
value="org.postgresql.xa.PGXADataSource"/>
      <param name="driver-module-name" value="org.postgresql.jdbc"/>
    </feature>
  </feature>
  <packages>
    <package name="org.postgresql.jdbc"/>
  </packages>
</layer-spec>
```

   - **postgresql-datasource**
     This directory contains a **layer-spec.xml** file with the following content:

```
<?xml version="1.0" ?>
<layer-spec xmlns="urn:jboss:galleon:layer-spec:1.0" name="postgresql-datasource">
  <dependencies>
    <layer name="postgresql-driver"/>
  </dependencies>
  <feature spec="subsystem.datasources.data-source">
    <param name="use-ccm" value="true"/>
    <param name="data-source" value="PostgreSQLDS"/>
    <param name="enabled" value="true"/>
    <param name="use-java-context" value="true"/>
    <param name="jndi-name"
value="java:jboss/datasources/${env.POSTGRESQL_DATASOURCE,env.OPENSHIFT_P
OSTGRESQL_DATASOURCE:PostgreSQLDS}"/>
    <param name="connection-url"
value="jdbc:postgresql://${env.POSTGRESQL_SERVICE_HOST,\
env.OPENSHIFT_POSTGRESQL_DB_HOST}:${env.POSTGRESQL_SERVICE_PORT,\

env.OPENSHIFT_POSTGRESQL_DB_PORT}/${env.POSTGRESQL_DATABASE,
env.OPENSHIFT_POSTGRESQL_DB_NAME}"/>
    <param name="driver-name" value="postgresql"/>
    <param name="user-name" value="${env.POSTGRESQL_USER,
env.OPENSHIFT_POSTGRESQL_DB_USERNAME}"/>
    <param name="password" value="${env.POSTGRESQL_PASSWORD,
env.OPENSHIFT_POSTGRESQL_DB_PASSWORD}"/>
    <param name="check-valid-connection-sql" value="SELECT 1"/>
```

```
        <param name="background-validation" value="true"/>
        <param name="background-validation-millis" value="60000"/>
        <param name="flush-strategy" value="IdleConnections"/>
        <param name="statistics-enabled" value="${wildfly.datasources.statistics-enabled:${wildfly.statistics-enabled:false}}" />
    </feature>
</layer-spec>
```

3. In the **pom.xml** file used to build your custom feature pack, refer to the JBoss EAP dependencies.

```
<dependency>
    <groupId>org.jboss.eap</groupId>
    <artifactId>wildfly-galleon-pack</artifactId>
    <version>2.0.0.GA-redhat-00002</version>
    <type>zip</type>
</dependency>
```

   These dependencies are available in the Red Hat Maven repository: https://maven.repository.redhat.com/ga/

4. Use the **build-user-feature-pack** goal in the Galleon Maven plugin to build custom layers.

Additional Resources

- Base Layers
- WildFly Galleon Maven Plugin Documentation
- Example illustrating packaging of drivers and datasources as Galleon layers

6.2.2. Custom provisioning files for JBoss EAP

Custom provisioning files are XML files with the file name **provisioning.xml** that are stored in the **galleon** subdirectory.

The following code illustrates a custom provisioning file:

```
<?xml version="1.0" ?>
<installation xmlns="urn:jboss:galleon:provisioning:3.0">
    <feature-pack location="eap-s2i@maven(org.jboss.universe:s2i-universe)"> 1
        <default-configs inherit="false"/> 2
        <packages inherit="false"/> 3
    </feature-pack>
    <feature-pack location="com.example.demo:my-galleon-feature-pack:1.0
"> 4
        <default-configs inherit="false"/>
        <packages inherit="false"/>
    </feature-pack>
    <config model="standalone" name="standalone.xml"> 5
        <layers>
            <include name="cloud-server"/>
            <include name="my-custom-driver"/>
            <include name="my-custom-datasource"/>
```

```
    </layers>
  </config>
  <options> (6)
    <option name="optional-packages" value="passive+"/>
  </options>
</installation>
```

1. This element instructs the provisioning process to provision the current eap-s2i feature-pack. Note that a builder image includes only one feature pack.

2. This element instructs the provisioning process to exclude default configurations.

3. This element instructs the provisioning process to exclude default packages.

4. This element instructs the provisioning process to provision the **com.example.demo:my-galleon-feature-pack:1.0** feature pack. The child elements instruct the process to exclude default configurations and default packages.

5. This element instructs the provisioning process to create a custom standalone configuration. The configuration includes the **cloud-server** base layer and the **my-custom-driver** and **my-custom-datasource** custom layers from the **com.example.demo:my-galleon-feature-pack:1.0** feature pack.

6. This element instructs the provisioning process to optimize provisioning of JBoss EAP modules.

6.2.3. Building an application provisioned with user-developed layers

When you build an application from a directory that includes a custom provisioning file, the S2I build process detects the provisioning file and provisions the JBoss EAP server as instructed.

Prerequisites

- The user-developed layers must exist in an accessible Maven repository.
- The application directory must contain a valid provisioning file that refers to the user-developed layers and the feature pack that contains them.

Procedure

- Enter a standard S2I build command to build the application.
  For example, assume you create the following custom provisioning file in your application directory.

```
<?xml version="1.0" ?>
<installation xmlns="urn:jboss:galleon:provisioning:3.0">
  <feature-pack location="eap-s2i@maven(org.jboss.universe:s2i-universe)">
    <default-configs inherit="false"/>
    <packages inherit="false"/>
  </feature-pack>
  <feature-pack location="com.example.demo:my-galleon-feature-pack:1.0">
    <default-configs inherit="false"/>
    <packages inherit="false"/>
  </feature-pack>
  <config model="standalone" name="standalone.xml">
```

```
    <layers>
        <include name="cloud-server"/>
        <include name="my-custom-driver"/>
        <include name="my-custom-datasource"/>
    </layers>
  </config>
  <options>
    <option name="optional-packages" value="passive+"/>
  </options>
</installation>
```

The following command builds an application using the **com.example.demo:my-galleon-feature-pack:1.0** feature pack, which includes the **my-custom-driver** and **my-custom-datasource** layers. The resulting application is named **eap-my-custom-db**. The connection to the database is configured using environment variables.

```
oc new-app --template=eap73-basic-s2i \
 -p APPLICATION_NAME=eap-my-custom-db \
 -p SOURCE_REPOSITORY_URL=https://github.com/<your repo>/eap-my-custom-db \
 -p SOURCE_REPOSITORY_REF=v1.0.0 \
 -p CONTEXT_DIR="" \
 -e DEMO_DB=demo \
 -e DEMO_PASSWORD=demo \
 -e DEMO_HOST=127.0.0.1  \
 -e DEMO_PORT=5432 \
 -e DEMO_USER=demo \
```

You can log in to the database on port 5432 with the user name **demo** and the password **demo**.

Additional Resources

Custom Provisioning Files for JBoss EAP

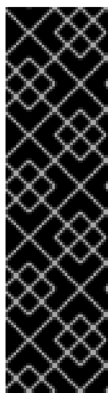
CHAPTER 7. ENABLE ECLIPSE MICROPROFILE APPLICATION DEVELOPMENT FOR JBOSS EAP ON RED HAT CODEREADY STUDIO

If you want to incorporate Eclipse MicroProfile capabilities in applications that you develop on CodeReady Studio, you must enable Eclipse MicroProfile support for JBoss EAP in CodeReady Studio.

JBoss EAP expansion packs provide support for Eclipse MicroProfile.

JBoss EAP expansion packs are not supported on JBoss EAP 7.2 and earlier.

Each version of the JBoss EAP expansion pack supports specific patches of JBoss EAP. For details, see the JBoss EAP expansion pack Support and Life Cycle Policies page.



IMPORTANT

The JBoss EAP XP Quickstarts for Openshift are provided as Technology Preview only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs), might not be functionally complete, and Red Hat does not recommend to use them for production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

See Technology Preview Features Support Scope on the Red Hat Customer Portal for information about the support scope for Technology Preview features.

7.1. CONFIGURING CODEREADY STUDIO TO USE ECLIPSE MICROPROFILE CAPABILITIES

To enable Eclipse MicroProfile support on JBoss EAP, register a new runtime server for JBoss EAP XP, and then create the new JBoss EAP 7.3 server.

Give the server an appropriate name that helps you recognize that it supports Eclipse MicroProfile capabilities.

This server uses a newly created JBoss EAP XP runtime that points to the runtime installed previously and uses the **standalone-microprofile.xml** configuration file.

Prerequisites

- JBoss EAP XP 2.0.0 has been installed .

Procedure

1. Set up the new server on the **New Server** dialog box.
    a. In the **Select server type** list, select *Red Hat JBoss Enterprise Application Platform 7.3* .
    b. In the **Server's host name** field, enter *localhost*.
    c. In the **Server name** field, enter *JBoss EAP 7.3 XP*.
    d. Click **Next**.

2. Configure the new server.

   a. In the **Home directory** field, if you do not want to use the default setting, specify a new directory; for example: *home/myname/dev/microprofile/runtimes/jboss-eap-7.3*.

   b. Make sure the **Execution Environment** is set to *JavaSE-1.8*.

   c. Optional: Change the values in the **Server base directory** and **Configuration file** fields.

   d. Click **Finish**.

Result

You are now ready to begin developing applications using Eclipse MicroProfile capabilities, or to begin using the Eclipse MicroProfile quickstarts for JBoss EAP.

7.2. USING ECLIPSE MICROPROFILE QUICKSTARTS FOR CODEREADY STUDIO

Enabling the Eclipse MicroProfile quickstarts makes the simple examples available to run and test on your installed server.

These examples illustrate the following Eclipse MicroProfile capabilities.

- Eclipse MicroProfile Config
- Eclipse MicroProfile Fault Tolerance
- Eclipse MicroProfile Health
- Eclipse MicroProfile JWT
- Eclipse MicroProfile Metrics
- Eclipse MicroProfile OpenAPI
- Eclipse MicroProfile OpenTracing
- Eclipse MicroProfile REST Client

Procedure

1. Import the **pom.xml** file from the Quickstart Parent Artifact.

2. If the quickstart you are using requires environment variables, configure the environment variables.
   Define environment variables on the launch configuration on the server **Overview** dialog box.

   For example, the **microprofile-opentracing** quickstart uses the following environment variables:

   - **JAEGER_REPORTER_LOG_SPANS** set to **true**
   - **JAEGER_SAMPLER_PARAM** set to **1**
   - **JAEGER_SAMPLER_TYPE** set to **const**

Additional resources

About Eclipse Microprofile

About JBoss Enterprise Application Platform expansion pack

Red Hat JBoss Enterprise Application Platform expansion pack Support and Life Cycle Policies

CHAPTER 8. THE BOOTABLE JAR

You can build and package a microservices application as a bootable JAR with the JBoss EAP JAR Maven plug-in. You can then run the application on a JBoss EAP bare-metal platform or a JBoss EAP OpenShift platform.

8.1. ABOUT THE BOOTABLE JAR

You can build and package a microservices application as a bootable JAR with the JBoss EAP JAR Maven plug-in.

A bootable JAR contains a server, a packaged application, and the runtime required to launch the server.

The JBoss EAP JAR Maven plug-in uses Galleon trimming capability to reduce the size and memory footprint of the server. Thus, you can configure the server according to your requirements, including only the Galleon layers that provide the capabilities that you need.

The JBoss EAP JAR Maven plug-in supports the execution of JBoss EAP CLI script files to customize your server configuration. A CLI script includes a list of CLI commands for configuring the server.

A bootable JAR is like a standard JBoss EAP server in the following ways:

- It supports JBoss EAP common management CLI commands.
- It can be managed using the JBoss EAP management console.

The following limitations exist when packaging a server in a bootable JAR:

- CLI management operations that require a server restart are not supported.
- The server cannot be restarted in admin-only mode, which is a mode that starts services related to server administration.
- If you shut down the server, updates that you applied to the server are lost.

Additionally, you can provision a hollow bootable JAR. This JAR contains only the server, so you can reuse the server to run a different application.

Additional resources

For information about capability trimming, see Capability Trimming.

8.2. JBOSS EAP MAVEN PLUG-IN

You can use the JBoss EAP JAR Maven plug-in to build an application as a bootable JAR.

You can retrieve the latest Maven plug-in version from in the Maven repository, which is available at Index of /ga/org/wildfly/plugins/wildfly-jar-maven-plugin..

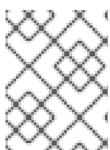
In a Maven project, the **src** directory contains all the source files required to build your application. After the JBoss EAP JAR Maven plug-in builds the bootable JAR, the generated JAR is located in **target/<application>-bootable.jar**.

The JBoss EAP JAR Maven plug-in also provides the following functionality:

- Applies CLI script commands to the server.
- Uses the **org.jboss.eap:wildfly-galleon-pack** Galleon feature pack and some of its layers for customizing the server configuration file.
- Supports the addition of extra files into the packaged bootable JAR, such as a keystore file.
- Includes the capability to create a hollow bootable JAR; that is, a bootable JAR that does not contain an application.

After you use the JBoss EAP JAR Maven plug-in to create the bootable JAR, you can start the application by issuing the following command. Replace **target/myapp-bootable.jar** with the path to your bootable JAR. For example:

```
$ java -jar target/myapp-bootable.jar
```



NOTE

To get a list of supported bootable JAR startup commands, append **--help** to the end of the startup command. For example, **java -jar target/myapp-bootable.jar --help**.

Additional resources

- For information about supported JBoss EAP Galleon layers, see Available JBoss EAP layers.
- For information about supported Galleon plug-ins to build feature packs for your project, see the WildFly Galleon Maven Plugin Documentation .
- For information about selecting methods to configure the JBoss EAP Maven repository, see Use the Maven Repository.
- For information about Maven project directories, see Introduction to the Standard Directory Layout in the *Apache Maven* documentation.

8.3. BOOTABLE JAR ARGUMENTS

View the arguments in the following table to learn about supported arguments for use with the bootable JAR.

Table 8.1. Supported bootable JAR executable arguments

Argument	Description
--help	Display the help message for the specified command and exit.
--deployment=<path>	Argument specific to the hollow bootable JAR. Specifies the path to the WAR, JAR, EAR file or exploded directory that contains the application you want to deploy on a server.

Argument	Description
--display-galleon-config	Print the content of the generated Galleon configuration file.
--install-dir=<path>	By default, the JVM settings are used to create a *TEMP* directory after the bootable JAR is started. You can use the **--install-dir** argument to specify a directory to install the server.
-secmgr	Runs the server with a security manager installed.
-b<interface>=<value>	Set system property **jboss.bind.address.<interface>** to the given value. For example, **bmanagement=IP_ADDRESS**.
-b=<value>	Set system property **jboss.bind.address**, which is used in configuring the bind address for the public interface. This defaults to 127.0.0.1 if no value is specified.
-D<name>[=<value>]	Specifies system properties that are set by the server at server runtime. The bootable JAR JVM does not set these system properties.
--properties=<url>	Loads system properties from a specified URL.
-S<name>[=value]	Set a security property.
-u=<value>	Set system property **jboss.default.multicast.address**, which is used in configuring the multicast address in the socket-binding elements in the configuration files. This defaults to 230.0.0.4 if no value is specified.

Argument	Description
--version	Display the application server version and exit.

8.4. SPECIFYING GALLEON LAYERS FOR YOUR BOOTABLE JAR SERVER

You can specify Galleon layers to build a custom configuration for your server. Additionally, you can specify Galleon layers that you want excluded from the server.

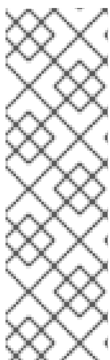
If you need to reference a single feature pack, use the **<feature-pack-location>** element to specify the location of the feature pack. The following example specifies **org.jboss.eap:wildfly-galleon-pack:2.0.0.GA-redhat-00002** in the **<feature-pack-location>** element of the Maven plug-in configuration file.

```
<configuration>
  <feature-pack-location>org.jboss.eap:wildfly-galleon-pack:2.0.0.GA-redhat-00002</feature-pack-location>
</configuration>
```

If you need to reference more than one feature pack, list them in the **<feature-packs>** element. The following example shows the addition of the Red Hat Single Sign-On feature pack to the **<feature-packs>** element:

```
<configuration>
    <feature-packs>
         <feature-pack>
             <location>org.jboss.eap:wildfly-galleon-pack:2.0.0.GA-redhat-00002</location>
        </feature-pack>
        <feature-pack>
            <location>org.jboss.sso:keycloak-adapter-galleon-pack:${version.keycloak}</location>
        </feature-pack>
    </feature-packs>
</configuration>
```

You can combine Galleon layers from multiple feature packs to configure the bootable JAR server to include only the supported Galleon layers that provide the capabilities that you need.



NOTE

On a bare-metal platform, if you do not specify Galleon layers in your configuration file then the provisioned server contains a configuration identical to that of a default **standalone-microprofile.xml** configuration.

After you add the **<cloud/>** configuration element in the plug-in configuration for the OpenShift platform, if you do not specify Galleon layers in your configuration file, the provisioned server contains a configuration that is adjusted for the cloud environment and is similar to a default **standalone-microprofile-ha.xml**.

Prerequisites

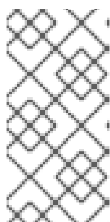
- Maven is installed.
- You have checked the latest Maven plug-in version. See Index of /ga/org/wildfly/plugins/wildfly-jar-maven-plugin.
- You have checked the latest Galleon feature-pack version, such as **2.0.X.GA-redhat-*BUILD_NUMBER***, where *X* is the minor version of JBoss EAP XP 2 and *BUILD_NUMBER* is the build number of the Galleon feature pack. Both *X* and *BUILD_NUMBER* can evolve during the JBoss EAP XP 2.0.0 product lifecycle. See link: Index of /ga/org/jboss/eap/wildfly-galleon-pack.

Procedure

1. Identify the supported JBoss EAP Galleon layers that provide the capabilities that you need to run your application.
2. Reference a JBoss EAP feature-pack location in the **<plugin>** element of the Maven project **pom.xml** file. You must specify the latest version of any Maven plug-in and the latest version of the **org.jboss.eap:wildfly-galleon-pack** Galleon feature pack, as demonstrated in the following example. The following example also displays the inclusion of a single feature-pack, which includes the **jaxrs-server** base layer and the **jpa-distributed** layer . This base layer provides additional support for the server:

```
<plugins>
    <plugin>
        <groupId>org.wildfly.plugins</groupId>
         <artifactId>wildfly-jar-maven-plugin</artifactId>
         <version>2.0.2.Final-redhat-00001</version>
        <configuration>
            <feature-pack-location>org.jboss.eap:wildfly-galleon-pack:2.0.0.GA-redhat-
00002</feature-pack-location>
            <layers>
               <layer>jaxrs-server</layer>
               <layer>jpa-distributed</layer>
            </layers>
            <excluded-layers>
               <layer>jpa</layer>
            </excluded-layers>
            ...
</plugins>
```

This example also shows the exclusion of the **jpa** layer from the project.



NOTE

If you include the **jpa-distributed** layer in your project, you must exclude the **jpa** layer from the **jaxrs-server** layer. The **jpa** layer configures a local infinispan hibernate cache, while the **jpa-distributed** layer configures a remote infinispan hibernate cache.

Additional resources

- For information about available base layers, see Base layers.

- For information about supported Galleon plug-ins to build feature packs for your project, see the WildFly Galleon Maven Plugin Documentation .
- For information about selecting methods to configure the JBoss EAP Maven repository, see Maven and the JBoss EAP Eclipse MicroProfile Maven repository .
- For information about managing your Maven dependencies, see Dependency Management in the *Apache Maven Project* documentation.

8.5. USING A BOOTABLE JAR ON A JBOSS EAP BARE-METAL PLATFORM

You can package an application as a bootable JAR on a JBoss EAP bare-metal platform.

A bootable JAR contains a server, a packaged application, and the runtime required to launch the server.

This procedure demonstrates packaging the MicroProfile Config microservices application as a bootable JAR with the JBoss EAP JAR Maven plug-in. See Eclipse MicroProfile Config development.

You can use CLI scripts to configure the server during the packaging of the bootable JAR.

IMPORTANT

On building a web application that must be packaged inside a bootable JAR, you must specify **war** in the **<packaging>** element of your **pom.xml** file. For example:

```
<packaging>war</packaging>
```

This value is required to package the build application as a WAR file and not as the default JAR file.

In a Maven project that is used solely to build a hollow bootable JAR, set the packaging value to **pom**. For example:

```
<packaging>pom</packaging>
```

Note that you are not limited to using **pom** packaging when building a hollow bootable JAR. You can create a hollow bootable JAR for a Maven project by specifying **true** in the **<hollow-jar>** element for any type of packaging, such as **war**. See Creating a hollow bootable JAR on a JBoss EAP bare-metal platform.

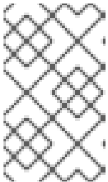
Prerequisites

- You have created a Maven project, setup a parent dependency, and added dependencies for creating a MicroProfile application. See Eclipse MicroProfile Config development.
- You have checked the latest Maven plug-in version. See Index of /ga/org/wildfly/plugins/wildfly-jar-maven-plugin.
- You have checked the latest Galleon feature-pack version, such as **2.0.X.GA-redhat-*BUILD_NUMBER***, where *X* is the minor version of JBoss EAP XP 2 and *BUILD_NUMBER* is the build number of the Galleon feature pack. Both *X* and *BUILD_NUMBER* can evolve during the JBoss EAP XP 2.0.0 product lifecycle. See link: Index of /ga/org/jboss/eap/wildfly-galleon-pack.

Procedure

1. Add the following content to the **<build>** element of the **pom.xml** file. You must specify the latest version of any Maven plug-in and the latest version of the **org.jboss.eap:wildfly-galleon-pack** Galleon feature pack. For example:

```
<plugins>
    <plugin>
        <groupId>org.wildfly.plugins</groupId>
        <artifactId>wildfly-jar-maven-plugin</artifactId>
        <version>2.0.2.Final-redhat-00001</version>
        <configuration>
             <feature-pack-location>org.jboss.eap:wildfly-galleon-pack:2.0.0.GA-redhat-
00002</feature-pack-location>
            <layers>
                <layer>jaxrs-server</layer>
                <layer>microprofile-platform</layer>
            </layers>
         </configuration>
        <executions>
            <execution>
                <goals>
                    <goal>package</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
```



NOTE

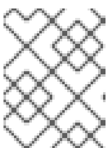
If you do not specify Galleon layers in your **pom.xml** file then the bootable JAR server contains a configuration that is identical to a **standalone-microprofile.xml** configuration.

2. Package the application as a bootable JAR:

```
$ mvn package
```

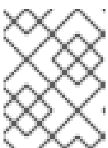
3. Start the application:

```
$ NAME="foo" java -jar target/microprofile-config-bootable.jar
```



NOTE

The example shows an environment variable called "NAME", but you can choose instead to use the default property value of "jim".



NOTE

To view a list of supported bootable JAR arguments, append **--help** to the end of the **java -jar target/microprofile-config-bootable.jar** command.

4. Specify the following URL in your web browser to access the Eclipse MicroProfile Config application:

```
http://localhost:8080/config/json
```

5. *Verification:* Test the application behaves properly by issuing the following command in your terminal:

```
curl http://localhost:8080/config/json
```

The following is the expected output:

```
{"result":"Hello foo"}
```

Additional resources

- For information about available Eclipse MicroProfile Config functionality, see Eclipse MicroProfile Config.
- For information about **ConfigSources**, see Eclipse MicroProfile Config reference.

8.6. CREATING A HOLLOW BOOTABLE JAR ON A JBOSS EAP BARE-METAL PLATFORM

You can package an application as a hollow bootable JAR on a JBoss EAP bare-metal platform.

A hollow bootable JAR contains only the JBoss EAP server. The hollow bootable JAR is packaged by the JBoss EAP JAR Maven plug-in. The application is provided at server runtime. The hollow bootable JAR is useful if you need to re-use the server configuration for a different application.

Prerequisites

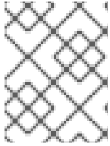
- You have created a Maven project, setup a parent dependency, and added dependencies for creating a MicroProfile application. See Develop Eclipse MicroProfile Applications for JBoss EAP.
- You have completed the **pom.xml** file configuration steps outlined in Using a bootable JAR on a JBoss EAP bare-metal platform.
- You have checked the latest Galleon feature-pack version, such as **2.0.X.GA-redhat-XXXXX**, where the value of *X* can change during the lifetime of JBoss EAP XP 2.0.0. See link: Index of /ga/org/jboss/eap/wildfly-galleon-pack.

Procedure

1. To build a hollow bootable JAR, you must set the **hollow-jar** plug-in configuration element to **true** in the project **pom.xml** file. For example:

```
<plugins>
    <plugin>
      ...
        <configuration>
          <!-- This example configuration does not show a complete plug-in configuration -->
```

```
            ...
            <feature-pack-location>org.jboss.eap:wildfly-galleon-pack:2.0.0.GA-redhat-00002</feature-
pack-location>
            <hollow-jar>true</hollow-jar>
        </configuration>
    </plugin>
</plugins>
```



NOTE

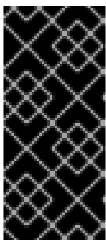
By specifying **true** in the **<hollow-jar>** element, the JBoss EAP JAR Maven plug-in does not include an application in the JAR.

1. Build the hollow bootable JAR:

```
$ mvn clean package
```

2. Run the hollow bootable JAR:

```
$ java -jar target/microprofile-config-bootable.jar --deployment=target/microprofile-config.war
```



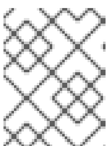
IMPORTANT

To specify the path to the WAR file that you want to deploy on the server, use the following argument, where **PATH_NAME** is the path to your deployment.

```
--deployment=<_PATH_NAME_>
```

3. Access the application:

```
$ curl http://localhost:8080/microprofile-config/config/json
```



NOTE

To register your web application in the root directory, name the application **ROOT.war**.

Additional resources

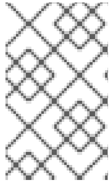
- For information about available Eclipse MicroProfile, see Eclipse MicroProfile Config.
- For more information about the JBoss EAP JAR Maven plug-in supported in JBoss EAP XP 2.0.0, see JBoss EAP Maven plug-in.

8.7. CLI SCRIPTS

You can create CLI scripts to configure the server during the packaging of the bootable JAR.

A CLI script is a text file that contains a sequence of CLI commands that you can use to apply additional server configurations. For example, you can create a script to add a new logger to the **logging** subsystem.

You can also specify more complex operations in a CLI script. For example, you can group security management operations into a single command to enable HTTP authentication for the management HTTP endpoint.



NOTE

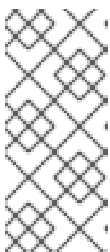
You must define CLI scripts in the **<cli-session>** element of the plug-in configuration before you package an application as a bootable JAR. This ensures the server configuration settings persist after packaging the bootable JAR.

Although you can combine predefined Galleon layers to configure a server that deploys your application, limitations do exist. For example, you cannot enable the HTTPS **undertow** listener using Galleon layers when packaging the bootable JAR. Instead, you must use a CLI script.

You must define the CLI scripts in the **<cli-session>** element of the **pom.xml** file. The following table shows types of CLI session attributes:

Table 8.2. CLI script attributes

Argument	Description
script-files	List of paths to script files.
properties-file	Optional attribute that specifies a path to a properties file. This file lists Java properties that scripts can reference by using the **${my.prop}** syntax. The following example sets **public inet-address** to the value of **all.addresses**: **/interface=public:write-attribute(name=inet-address,value=${all.addresses})**
resolve-expressions	Optional attribute that contains a boolean value. Indicates if system properties or expressions are resolved before sending the operation requests to the server. Value is **true** by default.

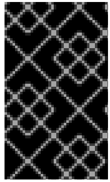


NOTE

- CLI scripts are started in the order that they are defined in the **<cli-session>** element of the **pom.xml** file.
- The JBoss EAP JAR Maven plug-in starts the embedded server for each CLI session. Thus, your CLI script does not have to start or stop the embedded server.

8.8. USING A BOOTABLE JAR ON A JBOSS EAP OPENSHIFT PLATFORM

After you packaged an application as a bootable JAR, you can run the application on a JBoss EAP OpenShift platform.



IMPORTANT

On OpenShift, you cannot use the EAP Operator automated transaction recovery feature with your bootable JAR. A fix for this technical limitation is planned for a future JBoss EAP XP 2.0.0 patch release.

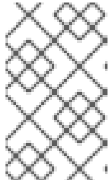
Prerequisites

- You have created a Maven project for Eclipse MicroProfile Config development.
- You have checked the latest Maven plug-in version. See Index of /ga/org/wildfly/plugins/wildfly-jar-maven-plugin.
- You have checked the latest Galleon feature-pack version, such as **2.0.X.GA-redhat-*BUILD_NUMBER***, where *X* is the minor version of JBoss EAP XP 2 and *BUILD_NUMBER* is the build number of the Galleon feature pack. Both *X* and *BUILD_NUMBER* can evolve during the JBoss EAP XP 2.0.0 product lifecycle. See link: Index of /ga/org/jboss/eap/wildfly-galleon-pack.

Procedure

1. Add the following content to the **<build>** element of the **pom.xml** file. You must specify the latest version of any Maven plug-in and the latest version of the **org.jboss.eap:wildfly-galleon-pack** Galleon feature pack. For example:

```
<plugins>
   <plugin>
      <groupId>org.wildfly.plugins</groupId>
      <artifactId>wildfly-jar-maven-plugin</artifactId>
      <version>2.0.2.Final-redhat-00001</version>
      <configuration>
         <feature-pack-location>org.jboss.eap:wildfly-galleon-pack:2.0.0.GA-redhat-
00002</feature-pack-location>
         <layers>
            <layer>jaxrs-server</layer>
            <layer>microprofile-platform</layer>
         </layers>
         <cloud/>
         </configuration>
         <executions>
            <execution>
               <goals>
                  <goal>package</goal>
               </goals>
            </execution>
         </executions>
   </plugin>
</plugins>
```



NOTE

You must include the **<cloud/>** element in the **<configuration>** element of the plug-in configuration, so the JBoss EAP Maven JAR plug-in can identify that you choose the OpenShift platform.

2. Package the application:

```
$ mvn package
```

3. Log in to your OpenShift instance using the **oc login** command.

4. Create a new project in OpenShift. For example:

```
$ oc new-project bootable-jar-project
```

5. Enter the following **oc** commands to create an application image:

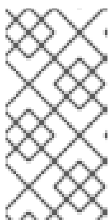
```
$ mkdir target/openshift && cp target/microprofile-config-bootable.jar target/openshift (1)

$ oc import-image ubi8/openjdk-11 --from=registry.redhat.io/ubi8/openjdk-11 --confirm (2)

$ oc new-build --strategy source --binary --image-stream openjdk-11 --name microprofile-config-app (3)

$ oc start-build microprofile-config-app --from-dir target/openshift (4)
```

1. Creates an openshift sub-directory in the target directory. The packaged application is copied into the created sub-directory.

2. Imports the latest OpenJDK 11 imagestream tag and image information into the OpenShift project.

3. Creates a build configuration based on the microprofile-config-app directory and the OpenJDK 11 imagestream.

4. Uses the **target/openshift** sub-directory as the binary input to build the application.



NOTE

OpenShift applies a set of CLI script commands to the bootable JAR configuration file to adjust it to the cloud environment. You can access this script by opening the **bootable-jar-build-artifacts/generated-cli-script.txt** file in the Maven project /**target directory**.

6. *Verification:*
View a list of OpenShift pods available and check the pods build statuses by issuing the following command:

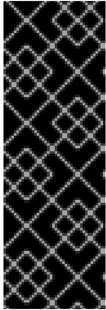
```
$ oc get pods
```

Verify the built application image:

```
$ oc get is microprofile-config-app
```

The output shows the built application image details, such as name and image repository, tag, and so on. For the example in this procedure, the imagestream name and tag output displays **microprofile-config-app:latest**.

7. Deploy the application.



IMPORTANT

To provide system properties to the bootable JAR, you must use the **JAVA_OPTS_APPEND** environment variable. The following example demonstrates usage of the **JAVA_OPTS_APPEND** environment variable.

```
oc new-app <_IMAGESTREAM_> -e JAVA_OPTS_APPEND="-
Xlog:gc*:file=/tmp/gc.log:time -Dwildfly.statistics-enabled=true"
```

```
$ oc new-app microprofile-config-app

$ oc expose svc/microprofile-config-app
```

A new application is created and started. The application configuration is exposed as a new service.

8. *Verification*: Test the application behaves properly by issuing the following command in your terminal:

```
$ curl http://$(oc get route microprofile-config-app --template='{{ .spec.host }}')/config/json
```

Expected output:

```
{"result":"Hello jim"}
```

Additional resources

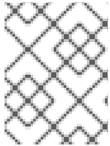
- For information about Eclipse MicroProfile, see Eclipse MicroProfile Config.
- For information about **ConfigSources**, see Default Eclipse MicroProfile Config attributes.

8.9. CONFIGURE THE BOOTABLE JAR FOR OPENSHIFT

Before using your bootable JAR, you can configure JVM settings to ensure that your standalone server operates correctly on JBoss EAP for OpenShift.

Use the **JAVA_OPTS_APPEND** environment variable to configure JVM settings. Use the **JAVA_ARGS** command to provide arguments to the bootable JAR.

You can use environment variables to set values for properties. For example, you can use the **JAVA_OPTS_APPEND** environment variable to set the **-Dwildfly.statistics-enabled** property to **true**. This enables statistics for your server.



NOTE

Use the **JAVA_ARGS** environment variable, if you need to provide arguments to the bootable JAR.

JBoss EAP for OpenShift provides a JDK 11 image. To run the application associated with your bootable JAR, you must first import the latest OpenJDK 11 imagestream tag and image information into your OpenShift project. You can then use environment variables to configure the JVM in the imported image.

You can apply the same configuration options for configuring the JVM used for JBoss EAP for OpenShift S2I image, but with the following differences:

- Optional: The **-Xlog** capability is not available, but you can set garbage collection logging by enabling **-Xlog:gc**. For example: **JAVA_OPTS_APPEND="-Xlog:gc*:file=/tmp/gc.log:time"**.
- To increase initial metaspace size, you can set the **GC_METASPACE_SIZE** environment variable. For best metadata capacity performance, set the value to **96**.
- The default value for **GC_MAX_METASPACE_SIZE** is set as **100**, but for best metadata capacity after a garbage collection, you must set it to at least **256**.
- For better random file generation, use the **JAVA_OPTS_APPEND** environment variable to set **java.security.egd** property as **-Djava.security.egd=file:/dev/urandom**. For example:

  ```
  JAVA_OPTS_APPEND="-Xlog:gc*:file=/tmp/gc.log:time -Dwildfly.statistics-enabled=true"
  ```

  These configurations improve the memory settings and garbage collection capability of JVM when running on your imported OpenJDK 11 image.

8.10. USING A CONFIGMAP IN YOUR APPLICATION ON OPENSHIFT

For OpenShift, you can use a deployment controller (dc) to mount the configmap into the pods used to run the application.

A **ConfigMap** is an OpenShift resource that is used to store non-confidential data in key-value pairs.

After you specify the **microprofile-platform** Galleon layer to add **microprofile-config-subsystem** and any extensions to the server configuration file, you can use a CLI script to add a new **ConfigSource** to the server configuration. You can save CLI scripts in an accessible directory, such as the **/scripts directory** in the root directory of your Maven project.

Eclipse MicroProfile Config functionality is implemented in JBoss EAP using the SmallRye Config component and is provided by the **microprofile-config-smallrye** subsystem. This subsystem is included in the **microprofile-platform** Galleon layer.

Prerequisites

- You have installed Maven.
- You have configured the JBoss EAP Maven repository.
- You have packaged an application as a bootable JAR and you can run the application on a JBoss EAP OpenShift platform. For information about building an application as a bootable JAR on an OpenShift platform, see Using a bootable JAR on a JBoss EAP OpenShift platform .

Procedure

1. Create a directory named **scripts** at the root directory of your project. For example:

```
$ mkdir scripts
```

2. Create a **cli.properties** file and save the file in the **/scripts** directory. Define the **config.path** and the **config.ordinal** system properties in this file. For example:

```
config.path=/etc/config
config.ordinal=200
```

3. Create a CLI script, such as **mp-config.cli**, and save it in an accessible directory in the bootable JAR, such as the /**scripts** directory. The following example shows the contents of the **mp-config.cli** script:

```
# config map

/subsystem=microprofile-config-smallrye/config-source=os-map:add(dir=
{path=${config.path}}, ordinal=${config.ordinal})
```

   The **mp-config.cli** CLI script creates a new **ConfigSource**, to which ordinal and path values are retrieved from a properties file.

4. Save the script in the /**scripts** directory, which is located at the root directory of the project.

5. Add the following configuration extract to the existing plug-in **<configuration>** element:

```
<cli-sessions>
    <cli-session>
        <properties-file>
            scripts/cli.properties
        </properties-file>
        <script-files>
            <script>scripts/mp-config.cli</script>
        </script-files>
    </cli-session>
</cli-sessions>
```

6. Package the application:

```
$ mvn package
```

7. Log in to your OpenShift instance using the **oc login** command.

8. *Optional:* If you have not previously created a **target/openshift** subdirectory, you must create the suddirectory by issuing the following command:

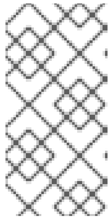
```
$ mkdir target/openshift
```

9. Copy the packaged application into the created subdirectory.

```
$ cp target/microprofile-config-bootable.jar target/openshift
```

10. Use the **target/openshift** subdirectory as the binary input to build the application:

```
$ oc start-build microprofile-config-app --from-dir target/openshift
```
(1)



NOTE

OpenShift applies a set of CLI script commands to the bootable JAR configuration file to adjust it to the cloud environment. You can access this script by opening the **bootable-jar-build-artifacts/generated-cli-script.txt** file in the Maven project /**target directory**.

11. Create a **ConfigMap**. For example:

```
$ oc create configmap microprofile-config-map --from-literal=name="Name comes from
Openshift ConfigMap"
```

12. Mount the **ConfigMap** into the application with the dc. For example:

```
$ oc set volume deployments/microprofile-config-app --add --name=config-volume \
--mount-path=/etc/config \
--type=configmap \
--configmap-name=microprofile-config-map
```

After executing the **oc set volume** command, the application is re-deployed with the new configuration settings.

13. Test the output:

```
$ curl http://$(oc get route microprofile-config-app --template='{{ .spec.host }}')/config/json
```

The following is the expected output:

```
{"result":"Hello Name comes from Openshift ConfigMap"}
```

Additional resources

- For information about Eclipse MicroProfile Config **ConfigSources** attributes, see Default Eclipse MicroProfile Config attributes.
- For information about bootable JAR arguments, see Supported bootable JAR arguments .

8.11. ENABLE WEB-SESSION DATA STORAGE FOR MULTIPLE BOOTABLE JAR INSTANCES

You can build and package a web-clustering application as a bootable JAR.

Prerequisites

- You have checked the latest Maven plug-in version. See Index of /ga/org/wildfly/plugins/wildfly-jar-maven-plugin.
- You have checked the latest Galleon feature-pack version, such as **2.0.X.GA-**

redhat-*BUILD_NUMBER*, where *X* is the minor version of JBoss EAP XP 2 and *BUILD_NUMBER* is the build number of the Galleon feature pack. Both *X* and *BUILD_NUMBER* can evolve during the JBoss EAP XP 2.0.0 product lifecycle. See link: Index of /ga/org/jboss/eap/wildfly-galleon-pack.

Procedure

1. Set up the Maven project. For example:

```
$ mvn archetype:generate \
-DgroupId=com.example.webclustering \
-DartifactId=web-clustering \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-webapp \
-DinteractiveMode=false
cd web-clustering
```

2. In the **pom.xml** file, configure the Maven repository to retrieve the JBoss EAP BOM file.

```
<repositories>
        <repository>
            <id>jboss</id>
            <url>https://maven.repository.redhat.com/ga</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>
```

3. To configure the **pom.xml** file to automatically manage versions for the Jakarta EE artifacts in the **jboss-eap-jakartaee8** BOM, import the BOM to the **<dependencyManagement>** section of the project **pom.xml** file. For example:

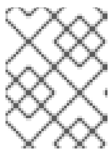
```
<dependencyManagement>
  <dependencies>
    <dependency>
        <groupId>org.jboss.bom</groupId>
        <artifactId>jboss-eap-jakartaee8</artifactId>
        <version>7.3.4.GA</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

4. Add the servlet API artifact, which is managed by the BOM, to the **<dependency>** section of the project **pom.xml** file. The following example demonstrates adding the servlet API dependency to the file:

```
<dependency>
    <groupId>org.jboss.spec.javax.servlet</groupId>
    <artifactId>jboss-servlet-api_4.0_spec</artifactId>
    <scope>provided</scope>
</dependency>
```

5. Add the following content to the **<build>** element of the **pom.xml** file. You must specify the latest version of any Maven plug-in and the latest version of the **org.jboss.eap:wildfly-galleon-pack** Galleon feature pack. For example:

```
<plugins>
   <plugin>
      <groupId>org.wildfly.plugins</groupId>
      <artifactId>wildfly-jar-maven-plugin</artifactId>
      <version>2.0.2.Final-redhat-00001</version>
      <configuration>
         <feature-pack-location>org.jboss.eap:wildfly-galleon-pack:2.0.0.GA-redhat-
00002</feature-pack-location>
         <layers>
            <layer>datasources-web-server</layer>
            <layer>web-clustering</layer>
         </layers>
      </configuration>
      <executions>
         <execution>
            <goals>
               <goal>package</goal>
            </goals>
         </execution>
      </executions>
   </plugin>
</plugins>
```



NOTE

This example makes use of the **web-clustering** Galleon layer to enable web session sharing.

6. Update the **web.xml** file in the **src/main/webapp/WEB-INF** directory with the following configuration:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="4.0"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd">
   <distributable/>
</web-app>
```

The content pertained in the <distributable/> element directs the server that this servlet can be distributed on multiple servers.

7. Create the directory to store Java files:

```
$ mkdir -p APPLICATION_ROOT
/src/main/java/com/example/webclustering/
```

Where **APPLICATION_ROOT** is the directory containing the **pom.xml** configuration file for the application.

8. Create a Java file **MyServlet.java** with the following content and save the file in the ***APPLICATION_ROOT*/src/main/java/com/example/webclustering/** directory.

```
package com.example.webclustering;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns = {"/clustering"})
public class MyServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws IOException {
    response.setContentType("text/html;charset=UTF-8");
    long t;
    User user = (User) request.getSession().getAttribute("user");
    if (user == null) {
      t = System.currentTimeMillis();
      user = new User(t);
      request.getSession().setAttribute("user", user);
    }
    try (PrintWriter out = response.getWriter()) {
      out.println("<!DOCTYPE html>");
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Web clustering demo</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h1>Session id " + request.getSession().getId() + "</h1>");
      out.println("<h1>User Created " + user.getCreated() + "</h1>");
      out.println("<h1>Host Name " + System.getenv("HOSTNAME") + "</h1>");
      out.println("</body>");
      out.println("</html>");
    }
  }
}
```

The content in **MyServlet.java** defines the endpoint to which a client sends an HTTP request.

9. Create a Java file **User.java** with the following content and save the file in the ***APPLICATION_ROOT*/src/main/java/com/example/webclustering/** directory.

```
package com.example.webclustering;

import java.io.Serializable;

public class User implements Serializable {
  private final long created;
```

```
    User(long created) {
        this.created = created;
    }
    public long getCreated() {
        return created;
    }
}
```

10. Package the application:

```
$ mvn package
```

11. *Optional*: To run the application on a JBoss EAP bare-metal platform, follow the steps outlined in Using a bootable JAR on a JBoss EAP bare-metal platform , but with the following difference:

    a. On a JBoss EAP bare-metal platform, you can use the **java -jar** command to run multiple bootable JAR instances, as demonstrated in the following examples:

    ```
    $ java -jar target/web-clustering-bootable.jar -Djboss.node.name=node1

    $ java -jar target/web-clustering-bootable.jar -Djboss.node.name=node2 -
    Djboss.socket.binding.port-offset=10
    ```

    b. *Verification*: You can access the application on the node 1 instance: http://127.0.0.1:8080/clustering. Note the user session ID and the user-creation time. After you kill this instance, you can access the node 2 instance: http://127.0.0.1:8090/clustering. The user must match the session ID and the user-creation time of the node 1 instance.

12. *Optional*: To run the application on a JBoss EAP OpenShift platform, follow the steps outlined in Using a bootable JAR on a JBoss EAP OpenShift platform , but complete the following steps:

    a. Add the **<cloud/>** element to the plug-in configuration. For example:

    ```
    <plugins>
      <plugin>
        ... <!-- You must evolve the existing configuration with the <cloud/> element  -->
        <configuration >
          ...
          <cloud/>
        </configuration>
      </plugin>
    </plugins>
    ```

    b. Re-build the application:

    ```
    $ mvn clean package
    ```

    c. Log in to your OpenShift instance using the **oc login** command.

    d. Create a new project in OpenShift. For example:

    ```
    $ oc new-project bootable-jar-project
    ```

e. To run a web-clustering application on a JBoss EAP OpenShift platform, authorization access must be granted for the service account that the pod is running in. The service account can then access the Kubernetes REST API. The following example shows authorization access being granted to a service account:

```
$ oc policy add-role-to-user view system:serviceaccount:$(oc project -q):default
```

f. Enter the following **oc** commands to create an application image:

```
$ mkdir target/openshift && cp target/web-clustering-bootable.jar target/openshift (1)

$ oc import-image ubi8/openjdk-11 --from=registry.redhat.io/ubi8/openjdk-11 --confirm
(2)

$ oc new-build --strategy source --binary --image-stream openjdk-11 --name web-
clustering (3)

$ oc start-build web-clustering --from-dir target/openshift (4)
```

(1) Creates the **target/openshift** sub-directory. The packaged application is copied into the **openshift** sub-directory.

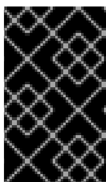
(1) (2) Imports the latest OpenJDK 11 imagestream tag and image information into the OpenShift project.

(3) Creates a build configuration based on the web-clustering directory and the OpenJDK 11 imagestream.

(4) Uses the **target/openshift** sub-directory as the binary input to build the application.

g. Deploy the application:

```
$ oc new-app web-clustering -e KUBERNETES_NAMESPACE=$(oc project -q)

$ oc expose svc/web-clustering
```



IMPORTANT

You must use the **KUBERNETES_NAMESPACE** environment variable to view other pods in the current OpenShift namespace; otherwise, the server attempts to retrieve the pods from the **default** namespace.

h. Get the URL of the route.

```
$ oc get route web-clustering --template='{{ .spec.host }}'
```

i. Access the application in your web browser using the URL returned from the previous command. For example:

```
http://ROUTE_NAME/clustering
```

Note the user session ID and user creation time.

j. Scale the application to two pods:

```
$ oc scale --replicas=2 deployments web-clustering
```

k. Issue the following command to view a list of OpenShift pods available and check the pods build statuses::

```
$ oc get pods
```

l. Kill the oldest pod using the **oc delete pod web-clustering-*POD_NAME*** command, where *POD_NAME* is the name of your oldest pod.

m. Access the application again:

```
http://ROUTE_NAME/clustering
```

Expected outcome: The session ID and the creation time generated by the new pod match those of the of the terminated pod. This indicates that web-session data storage is enabled.

Additional resources

- For information about distributable web-session management profiles, see The distributable-web subsystem for Distributable Web Session Configurations in the *Development Guide*.
- For information about configuring the JGroups protocol stack, see Configuring a JGroups Discovery Mechanism in the *Getting Started with JBoss EAP for OpenShift Container Platform* guide.

8.12. ENABLING HTTP AUTHENTICATION FOR BOOTABLE JAR WITH A CLI SCRIPT

You can enable HTTP authentication for the bootable JAR with a CLI script. This script adds a security realm and a security domain to your server.

Prerequisites

- You have checked the latest Maven plug-in version. See Index of /ga/org/wildfly/plugins/wildfly-jar-maven-plugin.
- You have checked the latest Galleon feature-pack version, such as **2.0.X.GA-redhat-*BUILD_NUMBER***, where *X* is the minor version of JBoss EAP XP 2 and *BUILD_NUMBER* is the build number of the Galleon feature pack. Both *X* and *BUILD_NUMBER* can evolve during the JBoss EAP XP 2.0.0 product lifecycle. See link: Index of /ga/org/jboss/eap/wildfly-galleon-pack.

Procedure

1. Set up the Maven project. For example:

```
$ mvn archetype:generate \
-DgroupId=com.example.auth \
-DartifactId=authentication \
-DarchetypeGroupId=org.apache.maven.archetypes \
```

```
-DarchetypeArtifactId=maven-archetype-webapp \
-DinteractiveMode=false
cd authentication
```

2. Configure the Maven repository in the **pom.xml** file to retrieve the JBoss EAP BOM file.

```
<repositories>
        <repository>
            <id>jboss</id>
            <url>https://maven.repository.redhat.com/ga</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>
```

3. To configure the **pom.xml** file to automatically manage versions for the Jakarta EE artifacts in the **jboss-eap-jakartaee8** BOM, import the BOM to the **<dependencyManagement>** section of the project **pom.xml** file. For example:

```
<dependencyManagement>
  <dependencies>
    <dependency>
        <groupId>org.jboss.bom</groupId>
        <artifactId>jboss-eap-jakartaee8</artifactId>
        <version>7.3.4.GA</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

4. Add the servlet API artifact, which is managed by the BOM, to the **<dependency>** section of the project **pom.xml** file. The following example demonstrates adding the servlet API dependency to the file:

```
<dependency>
    <groupId>org.jboss.spec.javax.servlet</groupId>
    <artifactId>jboss-servlet-api_4.0_spec</artifactId>
    <scope>provided</scope>
</dependency>
```

5. Add the following content to the **<build>** element of the **pom.xml** file. You must specify the latest version of any Maven plug-in and the latest version of the **org.jboss.eap:wildfly-galleon-pack** Galleon feature pack. For example:

```
<plugins>
    <plugin>
         <groupId>org.wildfly.plugins</groupId>
         <artifactId>wildfly-jar-maven-plugin</artifactId>
         <version>2.0.2.Final-redhat-00001</version>
         <configuration>
              <feature-pack-location>org.jboss.eap:wildfly-galleon-pack:2.0.0.GA-redhat-
00002</feature-pack-location>
```

```
            <layers>
                <layer>datasources-web-server</layer>
            </layers>
        </configuration>
        <executions>
            <execution>
                <goals>
                    <goal>package</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
```

The example shows the inclusion of the **datasources-web-server** Galleon layer that contains the **elytron** subsystem.

6. Create the **web.xml** file in the **src/main/webapp/WEB-INF** directory. For example:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="4.0"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd">

    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>Example Realm</realm-name>
    </login-config>

</web-app>
```

7. Create the directory to store Java files:

```
$ mkdir -p APPLICATION_ROOT/src/main/java/com/example/authentication/
```

8. Create a Java file **TestServlet.java** with the following content and save the file in the **APPLICATION_ROOT/src/main/java/com/example/authentication/** directory.

```
package com.example.authentication;

import javax.servlet.annotation.HttpMethodConstraint;
import javax.servlet.annotation.ServletSecurity;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.io.PrintWriter;

@WebServlet(urlPatterns = "/hello")
@ServletSecurity(httpMethodConstraints = { @HttpMethodConstraint(value = "GET",
```

```
rolesAllowed = { "Users" }) })
public class TestServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
        PrintWriter writer = resp.getWriter();
        writer.println("Hello " + req.getUserPrincipal().getName());
        writer.close();
    }

}
```

9. Create a CLI script, such as **authentication.cli**, and save it in an accessible directory in the bootable JAR, such as the ***APPLICATION_ROOT*/scripts** directory. **APPLICATION_ROOT** is the root directory of your Maven project. The script must contain the following commands:

```
/subsystem=elytron/properties-realm=bootable-realm:add(users-properties={relative-
to=jboss.server.config.dir, path=bootable-users.properties, plain-text=true}, groups-
properties={relative-to=jboss.server.config.dir, path=bootable-groups.properties})
/subsystem=elytron/security-domain=BootableDomain:add(default-realm=bootable-realm,
permission-mapper=default-permission-mapper, realms=[{realm=bootable-realm, role-
decoder=groups-to-roles}])

/subsystem=undertow/application-security-domain=other:write-attribute(name=security-
domain, value=BootableDomain)
```

10. Add the following configuration extract to the plug-in **<configuration>** element:

```
<cli-sessions>
    <cli-session>
        <script-files>
            <script>scripts/authentication.cli</script>
        </script-files>
    </cli-session>
</cli-sessions>
```

This example shows the **authentication.cli** CLI script, which configures the default **undertow** security domain to the security domain defined for your server.

11. In the root directory of your Maven project create a directory to store the properties files that the JBoss EAP JAR Maven plug-in adds to the bootable JAR:

```
$ mkdir -p _APPLICATION_ROOT_/extra-content/standalone/configuration/
```

Where **APPLICATION_ROOT** is the directory containing the **pom.xml** configuration file for the application.

This directory stores files such as **bootable-users.properties** and **bootable-groups.properties** files.

The **bootable-users.properties** file contains the following content:

```
testuser=bootable_password
```

The **bootable-groups.properties** file contains the following content:

```
testuser=Users
```

12. Add the following **extra-content-content-dirs** element to the existing **<configuration>** element:

```
<extra-server-content-dirs>
        <extra-content>extra-content</extra-content>
</extra-server-content-dirs>
```

The **extra-content** directory contains the properties files.

13. Package the application as a bootable JAR.

```
$ mvn package
```

14. Start the application:

```
mvn wildfly-jar:run
```

15. Call the servlet, but do not specify credentials:

```
curl -v http://localhost:8080/hello
```

Expected output:

```
HTTP/1.1 401 Unauthorized
...
WWW-Authenticate: Basic realm="Example Realm"
```

16. Call the server and specify your credentials. For example:

```
$ curl -v -u testuser:bootable_password http://localhost:8080/hello
```

A HTTP 200 status is returned that indicates HTTP authentication is enabled for your bootable JAR. For example:

```
HTTP/1.1 200 OK
....
Hello testuser
```

Additional resources

- For information about enabling HTTP authentication for the undertow security domain, see Enable HTTP Authentication for Applications Using the CLI Security Command in the *How to Configure Server Security*.

8.13. SECURING YOUR JBOSS EAP BOOTABLE JAR APPLICATION WITH RED HAT SINGLE SIGN-ON

You can use the Galleon **keycloak-client-oidc** layer to install a version of a server that is provisioned with Red Hat Single Sign-On 7.4 OpenID Connect client adapters.

The **keycloak-client-oidc** layer provides Red Hat Single Sign-On OpenID Connect client adapters to your Maven project. This layer is included with the **keycloak-adapter-galleon-pack** Red Hat Single Sign-On feature pack.

You can add the **keycloak-adapter-galleon-pack** feature pack to your JBoss EAP Maven plug-in configuration and then add the **keycloak-client-oidc**. You can view Red Hat Single Sign-On client adapters that are compatible with JBoss EAP by visiting the Supported Configurations: Red Hat Single Sign-On 7.4 web page.

The example in this procedure shows you how to secure a JBoss EAP bootable JAR by using JBoss EAP features provided by the **keycloak-client-oidc** layer.

Prerequisites

- Maven is installed.
- You have checked the latest Maven plug-in version. See Index of /ga/org/wildfly/plugins/wildfly-jar-maven-plugin.
- You have checked the latest Galleon feature-pack version, such as **2.0.X.GA-redhat-*BUILD_NUMBER***, where *X* is the minor version of JBoss EAP XP 2 and *BUILD_NUMBER* is the build number of the Galleon feature pack. Both *X* and *BUILD_NUMBER* can evolve during the JBoss EAP XP 2.0.0 product lifecycle. See Index of /ga/org/jboss/eap/wildfly-galleon-pack.
- You have a Red Hat Single Sign-On server that is running on port 8090. See Starting the Red Hat Single Sign-On server.
- You have logged in to the Red Hat Single Sign-On Admin Console and created the following metadata:
  - A realm named **demo**.
  - A role named **Users**.
  - A user and password. You must assign a **Users** role to the user.
  - A **simple-webapp** client with the Root URL: **http://localhost:8080/simple-webapp/secured**.

Procedure

1. Set up the Maven project. For example:

```
$ mvn archetype:generate \
-DgroupId=com.example.keycloak \
-DartifactId=simple-webapp \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-webapp \
-DinteractiveMode=false
cd simple-webapp
```

2. Configure the Maven repository in the **pom.xml** file to retrieve the JBoss EAP BOM file and Red Hat Single Sign-On Galleon feature pack:

```
<repositories>
        <repository>
            <id>jboss</id>
            <url>https://maven.repository.redhat.com/ga</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
</repositories>
```

3. To configure the **pom.xml** file to automatically manage versions for the Jakarta EE artifacts in the **jboss-eap-jakartaee8** BOM, import the BOM to the **<dependencyManagement>** section of the project **pom.xml** file. For example:

```
<dependencyManagement>
  <dependencies>
    <dependency>
        <groupId>org.jboss.bom</groupId>
        <artifactId>jboss-eap-jakartaee8</artifactId>
        <version>7.3.4.GA</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

4. Add the servlet API artifact, which is managed by the BOM, to the **<dependency>** section of the project **pom.xml** file. The following example demonstrates adding the servlet API dependency to the file:

```
<dependency>
    <groupId>org.jboss.spec.javax.servlet</groupId>
    <artifactId>jboss-servlet-api_4.0_spec</artifactId>
    <scope>provided</scope>
</dependency>
```

5. Add the following content to the **<build>** element of the **pom.xml** file. You must specify the latest version of any Maven plug-in and the latest version of the **org.jboss.eap:wildfly-galleon-pack** Galleon feature pack. For example:

```
<plugins>
    <plugin>
        <groupId>org.wildfly.plugins</groupId>
        <artifactId>wildfly-jar-maven-plugin</artifactId>
        <version>2.0.2.Final-redhat-00001</version>
        <configuration>
            <feature-packs>
                <feature-pack>
                    <location>org.jboss.eap:wildfly-galleon-pack:2.0.0.GA-redhat-00002</location>
                </feature-pack>
                <feature-pack>
                        <location>org.jboss.sso:keycloak-adapter-galleon-pack:9.0.10.redhat-
```

```
00001</location>
                </feature-pack>
            </feature-packs>
            <layers>
                <layer>datasources-web-server</layer>
                <layer>keycloak-client-oidc</layer>
            </layers>
        </configuration>
        <executions>
            <execution>
                <goals>
                    <goal>package</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
```

The Maven plug-in provisions subsystems and modules that are required for deploying the web application.

The **keycloak-client-oidc** layer provides Red Hat Single Sign-On OpenID Connect client adapters to your project by using the **keycloak** subsystem and its dependencies to activate support for Red Hat Single Sign-On authentication. Red Hat Single Sign-On client adapters are libraries that secure applications and services with Red Hat Single Sign-On.

6. In the project **pom.xml** file, set the **<context-root>** to **false** in your plug-in configuration. This registers the application in the **simple-webapp** resource path. By default, the WAR file is registered under the root-context path.

```
<configuration>
    ...
     <context-root>false</context-root>
     ...
</configuration>
```

7. Create a CLI script, such as **configure-oidc.cli** and save it in an accessible directory in the bootable JAR, such as the ***APPLICATION_ROOT*/scripts** directory, where *APPLICATION_ROOT* is the root directory of your Maven project. The script must contain the following commands:

```
/subsystem=keycloak/secure-deployment=simple-webapp.war:add( \
    realm=demo, \
    resource=simple-webapp, \
    public-client=true, \
    auth-server-url=http://localhost:8090/auth/, \
    ssl-required=EXTERNAL)
```

This script defines the **secure-deployment=simple-webapp.war** resource in the **keycloak** subsystem. The **simple-webapp.war** resource is the name of the WAR file that is deployed in the bootable JAR.

8. In the project **pom.xml** file, add the following configuration extract to the existing plug-in **<configuration>** element:

```
<cli-sessions>
    <cli-session>
        <script-files>
            <script>scripts/configure-oidc.cli</script>
        </script-files>
    </cli-session>
</cli-sessions>
```

9. Update the **web.xml** file in the **src/main/webapp/WEB-INF** directory. For example:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   metadata-complete="false">

    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>Simple Realm</realm-name>
    </login-config>

</web-app>
```

10. *Optional:* Alternatively to steps 7 through 9, you can embed the server configuration in the web application by adding the **keycloak.json** descriptor to the **WEB-INF** directory of the web application. For example:

```
{
    "realm" : "demo",
    "resource" : "simple-webapp",
    "public-client" : "true",
    "auth-server-url" : "http://localhost:8090/auth/",
    "ssl-required" : "EXTERNAL"
}
```

You must then set the **<auth-method>** of the web application to **KEYCLOAK**. The following example code illustrates how to set the **<auth-method>**:

```
    <login-config>
        <auth-method>KEYCLOAK</auth-method>
        <realm-name>Simple Realm</realm-name>
    </login-config>
```

11. Create a Java file named **SecuredServlet.java** with the following content and save the file in the ***APPLICATION_ROOT*/src/main/java/com/example/securedservlet/** directory.

```
package com.example.securedservlet;

import java.io.IOException;
import java.io.PrintWriter;
import java.security.Principal;
```

```
import javax.servlet.ServletException;
import javax.servlet.annotation.HttpMethodConstraint;
import javax.servlet.annotation.ServletSecurity;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/secured")
@ServletSecurity(httpMethodConstraints = { @HttpMethodConstraint(value = "GET",
    rolesAllowed = { "Users" }) })
public class SecuredServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        try (PrintWriter writer = resp.getWriter()) {
            writer.println("<html>");
            writer.println("<head><title>Secured Servlet</title></head>");
            writer.println("<body>");
            writer.println("<h1>Secured Servlet</h1>");
            writer.println("<p>");
            writer.print(" Current Principal '");
            Principal user = req.getUserPrincipal();
            writer.print(user != null ? user.getName() : "NO AUTHENTICATED USER");
            writer.print("'");
            writer.println("    </p>");
            writer.println("  </body>");
            writer.println("</html>");
        }
    }
}
```

12. Package the application as a bootable JAR.

```
$ mvn package
```

13. Start the application:

```
$ java -jar target/simple-webapp-bootable.jar
```

14. Specify the following URL in your web browser to access the webpage secured with Red Hat Single Sign-On:

```
http://localhost:8080/simple-webapp/secured
```

15. Log in as a user from your Red Hat Single Sign-On realm.

16. *Verification:* Check that the webpage displays the following output:

```
Current Principal '<principal id>'
```

Additional resources

- For information about configuring the Red Hat Single Sign-On adapter subsystem, see JBoss EAP Adapter in the *Securing Applications and Services Guide*.
- For information about specifying the JBoss EAP JAR Maven for your project, see Specifying Galleon layers for your bootable JAR server.

8.14. PACKAGING A BOOTABLE JAR IN DEV MODE

The JBoss EAP JAR Maven plug-in **dev goal** provides **dev** mode, Development Mode, which you can use to enhance your application development process.

In **dev** mode, you do not need to rebuild the bootable JAR after you make changes to your application.

The workflow in this procedure demonstrates using **dev** mode to configure a bootable JAR.

Prerequisities

- Maven is installed.
- You have created a Maven project, setup a parent dependency, and added dependencies for creating a MicroProfile application. See Develop Eclipse MicroProfile Applications for JBoss EAP.
- You have specified the JBoss EAP JAR Maven plug-in in your Maven project **pom.xml** file.

Procedure

1. Build and start the bootable JAR in Development Mode:

   ```
   $ mvn wildfly-jar:dev
   ```

   In **dev** mode, the server deployment scanner is configured to monitor the **target/deployments** directory.

2. Prompt the JBoss EAP Maven Plug-in to build and copy your application to the **target/deployments** directory with the following command:

   ```
   $ mvn package -Ddev
   ```

   The server packaged inside the bootable JAR deploys the application stored in the **target/deployments** directory.

3. Modify the code in your application code.

4. Use the **mvn package -Ddev** to prompt the JBoss EAP Maven Plug-in to re-build your application and re-deploy it.

5. Stop the server. For example:

   ```
   $ mvn wildfly-jar:shutdown
   ```

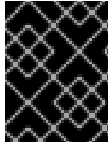
6. After you complete your application changes, package your application as a bootable JAR:

   ```
   $ mvn package
   ```

8.15. APPLYING THE JBOSS EAP PATCH TO YOUR BOOTABLE JAR

On a JBoss EAP bare-metal platform, you can install the patch to your bootable JAR by using a CLI script.

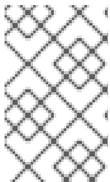
The CLI script issues the **patch apply** command to apply the patch during the bootable JAR build.



IMPORTANT

After you apply a patch to your bootable JAR, you cannot roll back from the applied patch. You must rebuild a bootable JAR without the patch.

Additionally, you can apply a legacy patch to your bootable JAR with the JBoss EAP JAR Maven plug-in. This plug-in provides a **<legacy-patch-cli-script>** configuration option to reference the CLI script that is used to patch the server.



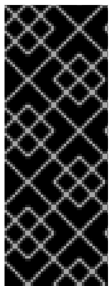
NOTE

The prefix **legacy-*** in **<legacy-patch-cli-script>** is related to applying archive patches to a bootable JAR. This method is similar to applying patches to regular JBoss EAP distributions.

You can use the **legacy-patch-cleanup** option in the JBoss EAP JAR Maven plug-in configuration to reduce the memory footprint of the bootable JAR by removing unused patch content. The option removes unused module dependencies. This option is set as **false** by default in the patch configuration file.

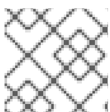
The **legacy-patch-cleanup** option removes the following patch content:

- The **<JBOSS_HOME>/.installation/patches** directory.
- Original locations of patch modules in the base layer.
- Unused modules that were added by the patch and are not referenced in the that existing module graph or patched modules graph.
- Overlays directories that are not listed in the **.overlays** file.



IMPORTANT

The **legacy-patch-clean-up** option variable is provided as a Technology Preview. Technology Preview features are not supported with Red Hat production service level agreements (SLAs), might not be functionally complete, and Red Hat does not recommend to use them for production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.



NOTE

The information outlined in this procedure also pertains to the hollow bootable JAR.

Prerequisites

- You have set up an account on the Red Hat Customer Portal.

- You have downloaded the following files from the **Product Downloads** page:
  - The JBoss EAP JBoss EAP 7.3.4 GA patch
  - The JBoss EAP XP 2.0.0 patch

Procedure

1. Create a CLI script that defines the legacy patches you want to apply to your bootable JAR. The script must contain one or more patch apply commands. The **--override-all** command is required when patching a server that was trimmed with Galleon layers, for example:

   ```
   patch apply patch-oneoff1.zip --override-all

   patch apply patch-oneoff2.zip --override-all

   patch info --json-output
   ```

2. Reference your CLI script in the **<legacy-patch-cli-script>** element of your **pom.xml** file.
3. Rebuild the bootable JAR.

Additional resources

- For information about downloading the JBoss EAP Eclipse MicroProfile Maven repository, see Downloading the JBoss EAP Eclipse MicroProfile Maven repository patch as an archive file .
- For information about creating CLI scripts, see CLI Scripts.
- For information about Technology Preview features, see Technology Preview Features Support Scope on the *Red Hat Customer Portal*.

CHAPTER 9. REFERENCE

9.1. ECLIPSE MICROPROFILE CONFIG REFERENCE

9.1.1. Default Eclipse MicroProfile Config attributes

The Eclipse MicroProfile Config specification defines three **ConfigSources** by default.

ConfigSources are sorted according to their ordinal number. If a configuration must be overwritten for a later deployment, the lower ordinal **ConfigSource** is overwritten before a higher ordinal **ConfigSource**.

Table 9.1. Default Eclipse MicroProfile Config attributes

ConfigSource	Ordinal
System properties	**400**
Environment variables	**300**
Property files **META-INF/microprofile-config.properties** found on the classpath	**100**

9.1.2. Eclipse MicroProfile Config SmallRye ConfigSources

The **microprofile-config-smallrye** project defines more **ConfigSources** you can use in addition to the default Eclipse MicroProfile Config **ConfigSources**.

Table 9.2. Additional Eclipse MicroProfile Config attributes

ConfigSource	Ordinal
config-source in the Subsystem	**100**
ConfigSource from the Directory	**100**
ConfigSource from Class	**100**

An explicit ordinal is not specified for these **ConfigSources**. They inherit the default ordinal value found in the Eclipse MicroProfile Config specification.

9.2. ECLIPSE MICROPROFILE FAULT TOLERANCE REFERENCE

9.2.1. Eclipse MicroProfile Fault Tolerance configuration properties

SmallRye Fault Tolerance specification defines the following properties in addition to the properties defined in the Eclipse MicroProfile Fault Tolerance specification.

Table 9.3. Eclipse MicroProfile Fault Tolerance configuration properties

Property	Default value	Description
io.smallrye.faulttolerance.globalThreadPoolSize	**100**	Number of threads used by the fault tolerance mechanisms. This does not include bulkhead thread pools.
io.smallrye.faulttolerance.timeoutExecutorThreads	**5**	Size of the thread pool used for scheduling timeouts.

9.3. ECLIPSE MICROPROFILE JWT REFERENCE

9.3.1. Eclipse MicroProfile Config JWT standard properties

The **microprofile-jwt-smallrye** subsystem supports the following Eclipse MicroProfile Config standard properties.

Table 9.4. Eclipse MicroProfile Config JWT standard properties

Property	Default	Description
mp.jwt.verify.publickey	NONE	String representation of the public key encoded using one of the supported formats. Do not set if you have set **mp.jwt.verify.publickey.location**.
mp.jwt.verify.publickey.location	NONE	The location of the public key, may be a relative path or URL. Do not be set if you have set **mp.jwt.verify.publickey**.
mp.jwt.verify.issuer	NONE	The expected value of any **iss** claim of any JWT token being validated.

Example **microprofile-config.properties** configuration:

```
mp.jwt.verify.publickey.location=META-INF/public.pem
mp.jwt.verify.issuer=jwt-issuer
```

9.4. ECLIPSE MICROPROFILE OPENAPI REFERENCE

9.4.1. Eclipse MicroProfile OpenAPI configuration properties

In addition to the standard Eclipse MicroProfile OpenAPI configuration properties, JBoss EAP supports the following additional Eclipse MicroProfile OpenAPI properties. These properties can be applied in both the global and the application scope.

Table 9.5. Eclipse MicroProfile OpenAPI properties in JBoss EAP

Property	Default value	Description
mp.openapi.extensions.enabled	**true**	Enables or disables registration of an OpenAPI endpoint. When set to **false**, disables generation of OpenAPI documentation. You can set the value globally using the config subsystem, or for each application in a configuration file such as **/META-INF/microprofile-config.properties**. You can parameterize this property to selectively enable or disable **microprofile-openapi-smallrye** in different environments, such as production or development. You can use this property to control which application associated with a given virtual host should generate a MicroProfile OpenAPI model.
mp.openapi.extensions.path	**/openapi**	You can use this property for generating OpenAPI documentation for multiple applications associated with a virtual host. Set a distinct **mp.openapi.extensions.path** on each application associated with the same virtual host.

Property	Default value	Description
mp.openapi.extensions.servers.relative	**true**	Indicates whether auto-generated server records are absolute or relative to the location of the OpenAPI endpoint. Server records are necessary to ensure, in the presence of a non-root context path, that consumers of an OpenAPI document can construct valid URLs to REST services relative to the host of the OpenAPI endpoint. The value **true** indicates that the server records are relative to the location of the OpenAPI endpoint. The generated record contains the context path of the deployment. When set to **false**, JBoss EAP XP generates server records including all the protocols, hosts, and ports at which the deployment is accessible.