# Red Hat JBoss Enterprise Application Platform 7.3

## Using Eclipse MicroProfile in JBoss EAP

For Use with JBoss EAP XP 1.0.0

Last Updated: 2020-09-22

# Red Hat JBoss Enterprise Application Platform 7.3 Using Eclipse MicroProfile in JBoss EAP

For Use with JBoss EAP XP 1.0.0

## Legal Notice

Copyright © 2020 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.

# Table of Contents

CHAPTER 1. JBOSS EAP XP FOR THE LATEST MICROPROFILE CAPABILITIES ..... 5
1.1. ABOUT JBOSS EAP XP 5
1.2. JBOSS EAP XP INSTALLATION 5
1.3. JBOSS EAP XP MANAGER FOR MANAGING JBOSS EAP XP PATCH STREAMS 5
1.4. INSTALLING JBOSS EAP XP 1.0.0 ON JBOSS EAP 7.3.0 6
1.5. INSTALLING JBOSS EAP XP 1.0.0 ON JBOSS EAP 7.3.1 6
1.6. UNINSTALLING JBOSS EAP XP 7
1.7. VIEWING THE STATUS OF JBOSS EAP XP 7

CHAPTER 2. UNDERSTAND ECLIPSE MICROPROFILE ..... 9
2.1. ECLIPSE MICROPROFILE CONFIG 9
2.1.1. Eclipse MicroProfile Config in JBoss EAP 9
2.1.2. Eclipse MicroProfile Config sources supported in Eclipse MicroProfile Config 9
2.2. ECLIPSE MICROPROFILE FAULT TOLERANCE 10
2.2.1. About Eclipse MicroProfile Fault Tolerance specification 10
2.2.2. Eclipse MicroProfile Fault Tolerance in JBoss EAP 10
2.3. ECLIPSE MICROPROFILE HEALTH 11
2.3.1. Eclipse MicroProfile Health in JBoss EAP 11
2.4. ECLIPSE MICROPROFILE JWT 12
2.4.1. Eclipse MicroProfile JWT integration in JBoss EAP 12
2.4.2. Differences between a traditional deployment and an Eclipse MicroProfile JWT deployment 12
2.4.3. Eclipse MicroProfile JWT activation in JBoss EAP 13
2.4.4. Limitations of Eclipse MicroProfile JWT in JBoss EAP 13
2.5. ECLIPSE MICROPROFILE METRICS 13
2.5.1. Eclipse MicroProfile Metrics in JBoss EAP 13
2.6. ECLIPSE MICROPROFILE OPENAPI 14
2.6.1. Eclipse MicroProfile OpenAPI in JBoss EAP 14
2.7. ECLIPSE MICROPROFILE OPENTRACING 14
2.7.1. Eclipse MicroProfile OpenTracing 14
2.7.2. Eclipse MicroProfile OpenTracing in EAP 14
2.8. ECLIPSE MICROPROFILE REST CLIENT 15
2.8.1. MicroProfile REST client 15

CHAPTER 3. ADMINISTER ECLIPSE MICROPROFILE IN JBOSS EAP ..... 17
3.1. ECLIPSE MICROPROFILE OPENTRACING ADMINISTRATION 17
3.1.1. Enabling MicroProfile Open Tracing 17
3.1.2. Removing the microprofile-opentracing-smallrye subsystem 17
3.1.3. Adding the microprofile-opentracing-smallrye subsystem 17
3.1.4. Installing Jaeger 18
3.2. ECLIPSE MICROPROFILE CONFIG CONFIGURATION 18
3.2.1. Adding properties in a ConfigSource management resource 18
3.2.2. Configuring directories as ConfigSources 18
3.2.3. Obtaining ConfigSource from a ConfigSource class 19
3.2.4. Obtaining ConfigSource configuration from a ConfigSourceProvider class 19
3.3. ECLIPSE MICROPROFILE FAULT TOLERANCE CONFIGURATION 20
3.3.1. Adding the MicroProfile Fault Tolerance extension 20
3.4. ECLIPSE MICROPROFILE HEALTH CONFIGURATION 21
3.4.1. Examining health using the management CLI 21
3.4.2. Examining health using the management console 21
3.4.3. Examining health using the HTTP endpoint 21
3.4.4. Enabling authentication for Eclipse MicroProfile Health 22

3.4.5. Global status when probes are not defined 22
3.5. ECLIPSE MICROPROFILE JWT CONFIGURATION 23
3.5.1. Enabling microprofile-jwt-smallrye subsystem 23
3.6. ECLIPSE MICROPROFILE METRICS ADMINISTRATION 23
3.6.1. Metrics available on the management interface 23
3.6.2. Examining metrics using the HTTP endpoint 24
3.6.3. Enabling Authentication for the Eclipse MicroProfile Metrics HTTP Endpoint 24
3.6.4. Obtaining the request count for a web service 24
3.7. ECLIPSE MICROPROFILE OPENAPI ADMINISTRATION 25
3.7.1. Enabling Eclipse MicroProfile OpenAPI 25
3.7.2. Requesting an Eclipse MicroProfile OpenAPI document using Accept HTTP header 26
3.7.3. Requesting an Eclipse MicroProfile OpenAPI document using an HTTP parameter 26
3.7.4. Configuring JBoss EAP to serve a static OpenAPI document 27
3.7.5. Disabling microprofile-openapi-smallrye 28
3.8. STANDALONE SERVER CONFIGURATION 28
3.8.1. Standalone server configuration files 28
3.8.2. Updating standalone configurations with Eclipse MicroProfile subsystems and extensions 30

CHAPTER 4. DEVELOP ECLIPSE MICROPROFILE APPLICATIONS FOR JBOSS EAP ..... 31
4.1. MAVEN AND THE JBOSS EAP ECLIPSE MICROPROFILE MAVEN REPOSITORY 31
4.1.1. Downloading the JBoss EAP Eclipse MicroProfile Maven repository patch as an archive file 31
4.1.2. Applying the JBoss EAP Eclipse MicroProfile Maven repository patch on your local system 31
4.1.3. Supported JBoss EAP Eclipse MicroProfile BOM 32
4.1.4. Using the JBoss EAP Eclipse MicroProfile Maven repository 33
4.2. ECLIPSE MICROPROFILE CONFIG DEVELOPMENT 34
4.2.1. Creating a Maven project for Eclipse MicroProfile Config 34
4.2.2. Using MicroProfile Config property in an application 35
4.3. ECLIPSE MICROPROFILE FAULT TOLERANCE APPLICATION DEVELOPMENT 37
4.3.1. Adding the MicroProfile Fault Tolerance extension 37
4.3.2. Configuring Maven project for Eclipse MicroProfile Fault Tolerance 38
4.3.3. Creating a fault tolerant application 39
4.4. ECLIPSE MICROPROFILE HEALTH DEVELOPMENT 42
4.4.1. Custom health check example 42
4.4.2. The @Liveness annotation example 43
4.4.3. The @Readiness annotation example 43
4.5. ECLIPSE MICROPROFILE JWT APPLICATION DEVELOPMENT 44
4.5.1. Enabling microprofile-jwt-smallrye subsystem 44
4.5.2. Configuring Maven project for developing JWT applications 44
4.5.3. Creating an application with Eclipse MicroProfile JWT 45
4.6. ECLIPSE MICROPROFILE METRICS DEVELOPMENT 50
4.6.1. Creating an Eclipse MicroProfile Metrics application 50
4.7. DEVELOPING AN ECLIPSE MICROPROFILE OPENAPI APPLICATION 52
4.7.1. Enabling Eclipse MicroProfile OpenAPI 52
4.7.2. Configuring Maven project for Eclipse MicroProfile OpenAPI 52
4.7.3. Creating an Eclipse MicroProfile OpenAPI application 54
4.7.4. Configuring JBoss EAP to serve a static OpenAPI document 58
4.8. ECLIPSE MICROPROFILE REST CLIENT DEVELOPMENT 59
4.8.1. A comparison between MicroProfile REST client and JAX-RS syntaxes 59
4.8.2. Programmatic registration of providers in MicroProfile REST client 60
4.8.3. Declarative registration of providers in MicroProfile REST client 60
4.8.4. Declarative specification of headers in MicroProfile REST client 60
4.8.5. Propagation of headers on the server in MicroProfile REST client 61
4.8.6. ResponseExceptionMapper in MicroProfile REST client 62

4.8.7. Context dependency injection with MicroProfile REST client 62

**CHAPTER 5. BUILD AND RUN MICROSERVICES APPLICATIONS ON THE OPENSHIFT IMAGE FOR JBOSS EAP XP 64**
5.1. PREPARING OPENSHIFT FOR APPLICATION DEPLOYMENT 64
5.2. CONFIGURING AUTHENTICATION TO THE RED HAT CONTAINER REGISTRY 65
5.3. IMPORTING THE LATEST OPENSHIFT IMAGE STREAMS AND TEMPLATES FOR JBOSS EAP XP 65
5.4. DEPLOYING A JBOSS EAP XP SOURCE-TO-IMAGE (S2I) APPLICATION ON OPENSHIFT 67
5.5. COMPLETING POST-DEPLOYMENT TASKS FOR JBOSS EAP XP SOURCE-TO-IMAGE (S2I) APPLICATION 69

**CHAPTER 6. ENABLE ECLIPSE MICROPROFILE APPLICATION DEVELOPMENT FOR JBOSS EAP ON RED HAT CODEREADY STUDIO 71**
6.1. INSTALLING THE JBOSS EAP XP IN CODEREADY STUDIO 71
6.2. CONFIGURING CODEREADY STUDIO TO USE ECLIPSE MICROPROFILE CAPABILITIES 72
6.3. USING ECLIPSE MICROPROFILE QUICKSTARTS FOR CODEREADY STUDIO 73

**CHAPTER 7. REFERENCE 75**
7.1. ECLIPSE MICROPROFILE CONFIG REFERENCE 75
7.1.1. Default Eclipse MicroProfile Config attributes 75
7.1.2. Eclipse MicroProfile Config SmallRye ConfigSources 75
7.2. ECLIPSE MICROPROFILE FAULT TOLERANCE REFERENCE 75
7.2.1. Eclipse MicroProfile Fault Tolerance configuration properties 75
7.3. ECLIPSE MICROPROFILE JWT REFERENCE 76
7.3.1. Eclipse MicroProfile Config JWT standard properties 76
7.4. ECLIPSE MICROPROFILE OPENAPI REFERENCE 76
7.4.1. Eclipse MicroProfile OpenAPI configuration properties 76

# 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 1.0.0 is compatible with JBoss EAP 7.3.1. If you want to install JBoss EAP XP 1.0.0 on JBoss EAP 7.3.0, you must first apply the JBoss EAP 7.3.1 GA patch.

**Additional Resources**

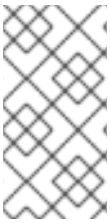
- Installing JBoss EAP XP 1.0.0 on JBoss EAP 7.3.0
- Installing JBoss EAP XP 1.0.0 on JBoss EAP 7.3.1

## 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.

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. INSTALLING JBOSS EAP XP 1.0.0 ON JBOSS EAP 7.3.0

JBoss JBoss EAP XP 1.0.0 is certified with JBoss EAP 7.3.1.

When you install JBoss EAP XP 1.0.0 on the JBoss EAP 7.3.0 server, you must apply a patch to upgrade it to JBoss EAP 7.3.1.

### Prerequisites

You have downloaded the following files from the **Product Downloads** page:

- The **jboss-eap-xp-1.0.0-manager.jar** file (JBoss EAP XP manager)
- JBoss EAP 7.3.1 GA patch
- The JBoss EAP XP 1.0.0 patch

### Procedure

1. Apply the JBoss EAP 7.3.1 GA patch using the following management command:

```
patch apply /path/to/jboss-eap-7.3.1-patch.zip
```

2. Set up JBoss EAP XP manager using the following command:

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

3. Apply the JBoss EAP XP 1.0.0 patch using the following management command:

```
patch apply /path/to/jboss-eap-xp-1.0.0-patch.zip
```

4. Restart the server:

```
shutdown --restart
```

## 1.5. INSTALLING JBOSS EAP XP 1.0.0 ON JBOSS EAP 7.3.1

### Prerequistes

You have downloaded the following files from the **Product Downloads** page:

- The **jboss-eap-xp-1.0.0-manager.jar** file (JBoss EAP XP manager)

- The JBoss EAP XP 1.0.0 patch

**Procedure**

1. Set up JBoss EAP XP manager using the following command:

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

2. Apply the JBoss EAP XP 1.0.0 patch using the following management command:

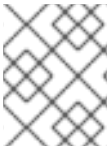
```
patch apply /path/to/jboss-eap-xp-1.0.0-patch.zip
```

3. Restart the server:

```
shutdown --restart
```

## 1.6. UNINSTALLING JBOSS EAP XP

Uninstalling JBoss EAP XP removes all the files related to enabling the JBoss EAP XP 1.0.0 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 1.0.0 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.

## 1.7. 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
- The available JBoss EAP XP manager commands to change the state
- Any support policy changes due to being in the current 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 1.0.0 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

By default, the **microprofile-health-smallrye** subsystem only examines whether the server is running. The **:empty-readiness-checks-status** and **: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 Jeager Client tracer using system properties and environment variables is provided as a Technology Preview. The system properties and environment variables affiliated with the Jeager 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 1.0.0 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. 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 1.0.0 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. |

| Configuration File | Purpose | Included capabilities | Excluded capabilities |
|---|---|---|---|
| **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:<br>• EJB<br>• Messaging<br>• Java Batch<br>• JavaServer Faces<br>• EJB Timers |
| **standalone-ha.xml** | | Includes default subsystems and adds the **modcluster** and **jgroups** subsystems for high availability. | Excludes subsystems necessary for messaging. |
| **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 **1.0.0**.
6. Click the **Releases** tab.
7. Find **JBoss EAP XP 1.0.0 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 1.0.0 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 1.0.0 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-1.0.0-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 1.0.0 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.

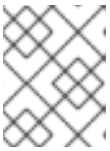
**Prerequisites**

- 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>1.0.0.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>1.0.0.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="h
elloworld-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.
  For information about configuring the JBoss EAP Maven repository, see Configuring the JBoss EAP Maven repository with the POM file.

### 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>1.0.0.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.

### 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:

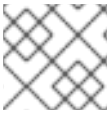
```
<fully qualified name of TestDataBase>/mp-rest/url=<URL>
```

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

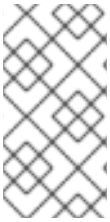
You can build and run your microservices applications on the OpenShift image for JBoss EAP XP.



**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 1.0.0 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 1.0.0 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 IMAGE STREAMS AND TEMPLATES FOR JBOSS EAP XP

Import the latest OpenShift image streams and templates for JBoss EAP XP.

**Procedure**

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

```
oc replace --force -f https://raw.githubusercontent.com/jboss-container-images/jboss-eap-openshift-templates/eap-xp1/jboss-eap-xp1-openjdk8-openshift.json
```

This command imports the following imagestreams and templates:

- The JDK 8 builder imagestream: jboss-eap-xp1-openjdk8-openshift
- The JDK 8 runtime imagestream: jboss-eap-xp1-openjdk8-runtime-openshift

b. Import JDK 11 image stream:

```
oc replace --force -f https://raw.githubusercontent.com/jboss-container-images/jboss-eap-openshift-templates/eap-xp1/jboss-eap-xp1-openjdk11-openshift.json
```

This command imports the following imagestreams and templates:

- The JDK 11 builder imagestream: jboss-eap-xp1-openjdk11-openshift
- The JDK 11 runtime imagestream: jboss-eap-xp1-openjdk11-runtime-openshift

c. Import the JDK 8 and JDK 11 templates:

```
for resource in \
eap-xp1-basic-s2i.json \
eap-xp1-third-party-db-s2i.json
do
oc replace --force -f https://raw.githubusercontent.com/jboss-container-images/jboss-eap-openshift-templates/eap-xp1/templates/${resource}
done
```

NOTE

The JBoss EAP XP image streams 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 image streams 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 image streams 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.

### 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-xp1-basic-s2i \ 1
 -p EAP_IMAGE_NAME=jboss-eap-xp1-openjdk8-openshift:1.0 \
 -p EAP_RUNTIME_IMAGE_NAME=jboss-eap-xp1-openjdk8-runtime-openshift:1.0 \
 -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-1.0.x \ 4
 -p CONTEXT_DIR=microprofile-config 5
```

1. The template to use.
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 image stream. 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-xp1-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-xp1-basic-s2i \ 1
 -p EAP_IMAGE_NAME=jboss-eap-xp1-openjdk11-openshift:1.0 \
 -p EAP_RUNTIME_IMAGE_NAME=jboss-eap-xp1-openjdk11-runtime-openshift:1.0 \
 -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-1.0.x \ 4
 -p CONTEXT_DIR=microprofile-config 5
```

1. The template to use.

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

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-xp1-basic-app-build-artifacts

…
Push successful
$ oc logs -f buildconfig/eap-xp1-basic-app
…
Push successful
```

**Additional resources**

- Importing the latest OpenShift image streams 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:

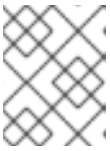
- Exposing a service so that the application is viewable from outside of OpenShift.
- Scaling your application to a specific number of replicas.

**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-xp1-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 1.0.0 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 JBoss EAP XP 1.0.1 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-xp1-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. 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.

## 6.1. INSTALLING THE JBOSS EAP XP IN CODEREADY STUDIO

You must install the JBoss EAP XP in CodeReady Studio to make the Eclipse MicroProfile capabilities available for application development.

**NOTE**

JBoss EAP expansion packs are not supported on JBoss EAP 7.2 and earlier.

**Prerequisites**

- Set up JBoss EAP 7.3 on CodeReady Studio.
- Download the following software artifacts:
  - The appropriate JBoss EAP 7.3 patch. See the Red Hat JBoss Enterprise Application Platform expansion pack Support and Life Cycle Policies page for information about the correct patches of JBoss EAP and JBoss EAP XP to install.
  - JBoss EAP XP manager.
  - The appropriate JBoss EAP XP patch. See the Red Hat JBoss Enterprise Application Platform expansion pack Support and Life Cycle Policies page for information about the correct patches of JBoss EAP and JBoss EAP XP to install.

**Procedure**

1. Navigate to the JBoss EAP installation directory that you specified when installing JBoss EAP through CodeReady Studio.
2. Apply the JBoss EAP patch that you downloaded earlier.

```
$ patch apply /_DOWNLOAD_/_PATH_/jboss-eap-<patch_id>-patch.zip
```

   For example:

```
$ patch apply /_DOWNLOAD_/_PATH_/jboss-eap-7.3.1-patch.zip
```

3. Set up JBoss EAP XP manager in your CodeReady Studio installation of JBoss EAP.

```
$ java -jar jboss-eap-xp-<patch_id>-manager.jar setup --jboss-home=/_PATH_/_TO_/_EAP_
```

   For example:

```
$ java -jar jboss-eap-xp-1.0.0.GA-CR1-manager.jar setup --jboss-
home=/_PATH_/_TO_/_EAP_
```

4. Apply the JBoss EAP XP patch that you downloaded.

```
$ patch apply /_DOWNLOAD_/_PATH_/jboss-eap-xp-<patch_id>-patch.zip
```

   For example:

```
$ patch apply /_DOWNLOAD_/_PATH_/jboss-eap-xp-1.0.0.GA-patch.zip
```

### Next steps

Configure CodeReady Studio to use MicroProfile capabilities

### Additional resources

- JBoss EAP product download page
- Red Hat JBoss Enterprise Application Platform expansion pack Support and Life Cycle Policies

## 6.2. 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 has been installed in CodeReady Studio .

**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.

## 6.3. 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 7. REFERENCE

## 7.1. ECLIPSE MICROPROFILE CONFIG REFERENCE

### 7.1.1. 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 7.1. 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** |

### 7.1.2. 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 7.2. 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.

## 7.2. ECLIPSE MICROPROFILE FAULT TOLERANCE REFERENCE

### 7.2.1. 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 7.3. 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. |

## 7.3. ECLIPSE MICROPROFILE JWT REFERENCE

### 7.3.1. Eclipse MicroProfile Config JWT standard properties

The **microprofile-jwt-smallrye** subsystem supports the following Eclipse MicroProfile Config standard properties.

**Table 7.4. 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
```

## 7.4. ECLIPSE MICROPROFILE OPENAPI REFERENCE

### 7.4.1. 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 7.5. Eclipse MicroProfile OpenAPI properties in JBoss EAP**

| Property | Default value | Description |
| --- | --- | --- |
| **mp.openapi.extensions.enabled** | **true** | Enables or disables registration of an OpenAPI endpoint.<br><br>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**.<br><br>You can parameterize this property to selectively enable or disable **microprofile-openapi-smallrye** in different environments, such as production or development.<br><br>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.<br><br>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.<br><br>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.<br><br>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.<br><br>When set to **false**, JBoss EAP XP generates server records including all the protocols, hosts, and ports at which the deployment is accessible. |