6

RED HAT JBOSS ENTERPRISE APPLICATION PLATFORM 8.0

USING JBOSS EAP XP 5.0

FOR USE WITH JBOSS EAP XP 5.0

Red Hat Customer Content Services

Abstract

This document provides general information about using JBoss EAP XP 5.0.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

PROVIDING FEEDBACK ON JBOSS EAP DOCUMENTATION

To report an error or to improve our documentation, log in to your Red Hat Jira account and submit an issue. If you do not have a Red Hat Jira account, then you will be prompted to create an account.

Procedure

1. Click the following link to **create a ticket**.
2. Enter a brief description of the issue in the **Summary**.
3. Provide a detailed description of the issue or enhancement in the **Description**. Include a URL to where the issue occurs in the documentation.
4. Clicking **Submit** creates and routes the issue to the appropriate documentation team.

CHAPTER 1. JBOSS EAP XP FOR THE LATEST MICROPROFILE CAPABILITIES

1.1. INSTALLING JBOSS EAP XP 5.0 WITHOUT A PRE-EXISTING JBOSS EAP 8.0 SERVER

If you have a JBoss EAP 7.4 installation and you want to install JBoss EAP XP 5.0 without first pre-installing JBoss EAP 8.0 server, follow the procedure below.

Prerequisites

- You have access to the internet.
- You have created an account on the Red Hat customer portal and are logged in.
- You have downloaded the **`jboss-eap-installation-manager`**.

Procedure

1. Open the terminal emulator and navigate to the **`jboss-eap-installation-manager`** directory.
2. Install JBoss EAP XP by running the following command from the **`jboss-eap-installation-manager`** directory :

```
./bin/jboss-eap-installation-manager.sh install --profile
eap-xp-5.0 --dir eap-xp-5
```

1.2. ADDING JBOSS EAP XP 5.0 FEATURE PACKS TO AN EXISTING JBOSS EAP 8.0 INSTALLATION

You can add an additional JBoss EAP XP 5.0 feature pack to an existing JBoss EAP installation using the **`jboss-eap-installation-manager`**.

Prerequisites

- You have an account on the Red Hat Customer Portal and are logged in.
- You have reviewed the supported configurations for JBoss EAP XP 5.0.
- You have installed a supported JDK.

- You have downloaded the **`jboss-eap-installation-manager`**. For more information about downloading **`jboss-eap-installation-manager`**, see the Installation Guide.
- You have downloaded or installed JBoss EAP 8.0 using one of the supported methods. For more information about downloading [ProductShortName], see the Installation Guide.

> **Note**
>
> Installing the JBoss EAP XP extension will automatically perform a server update to receive the latest component updates.

Procedure

1. Open the terminal emulator and navigate to the **`jboss-eap-installation-manager`** directory.
2. Run this script from the **`jboss-eap-installation-manager`** directory to subscribe the server to the JBoss EAP XP channel by executing:

```
./bin/jboss-eap-installation-manager.sh channel add \
 --channel-name eap-xp-5.0 \
 --repositories=mrrc-ga::https://
maven.repository.redhat.com/ga \
 --manifest org.jboss.eap.channels:eap-xp-5.0 \
 --dir eap-xp-5.0
```

3. Install JBoss EAP XP extension by executing:

```
./bin/jboss-eap-installation-manager.sh feature-pack add
\
 --fpl org.jboss.eap.xp:wildfly-galleon-pack \
 --dir eap-xp-5.0
```

1.3. ADDING JBOSS EAP XP 5.0 FEATURE PACKS TO AN EXISTING JBOSS EAP 8.0 INSTALLATION OFFLINE

You can add additional JBoss EAP XP 5.0 feature pack to an existing JBoss EAP installation offline using the **`jboss-eap-installation-manager`**.

Prerequisites

- You have reviewed the supported configurations for JBoss EAP XP 5.0.
- You have installed a supported JDK.

- You have downloaded the **`jboss-eap-installation-manager`**. For more information about downloading the **`jboss-eap-installation-manager`**, see the Installation Guide.
- You have downloaded or installed JBoss EAP 8.0 using one of the supported methods. For more information about downloading the JBoss EAP 8.0, see the Installation Guide.
- You have downloaded and extracted the latest offline repositories for JBoss EAP 8.0 and JBoss EAP XP 5.0.

Procedure

1. Open the terminal emulator and navigate to the **`jboss-eap-installation-manager`** directory.
2. Run this script from the **`jboss-eap-installation-manager`** directory to subscribe the server to the JBoss EAP XP channel by executing:

```
./bin/jboss-eap-installation-manager.sh channel add \
 --channel-name eap-xp-5.0 \
 --repositories=mrrc-ga::https://
maven.repository.redhat.com/ga \
 --manifest org.jboss.eap.channels:eap-xp-5.0 \
 --dir eap-xp-5.0
```

1. Install JBoss EAP XP and use the **`--repositories parameter`** to specify the offline repositories:

```
./bin/jboss-eap-installation-manager.sh feature-pack add \
 --fpl org.jboss.eap.xp:wildfly-galleon-pack \
 --dir eap-xp-5.0 \
 --repositories
<JBOSS_EAP_XP_OFFLINE_REPO_PATH>,<JBOSS_EAP_8.0_OFFLINE_REPO
_PATH>
```

Note

The feature pack will be added to the JBoss EAP installation passed in the **`--dir option`**.

1.4. UPDATING JBOSS EAP XP INSTALLATION USING THE JBOSS-EAP-INSTALLATION-MANAGER

You can update JBoss EAP XP periodically if new updates are available after you have downloaded and installed it.

Prerequisites

- You have access to the internet.

- You have installed a supported JDK.
- You have downloaded the **`jboss-eap-installation-manager`**. For more information about downloading **`jboss-eap-installation`**, see the Installation Guide.
- You have downloaded or installed JBoss EAP XP 5.0 using one of the supported methods. For more information, see the Installation Guide.

Procedure

1. Extract the **`jboss-eap-installation-manager`** you have downloaded.
2. Open the terminal emulator and navigate to the **`jboss-eap-installation-manager`** directory you have extracted.
3. Run this script from the **`jboss-eap-installation-manager`** directory to check for available updates:

```
./bin/jboss-eap-installation-manager.sh update list --dir
eap-xp-5.0
```

4. Update JBoss EAP by running the following command:

Syntax

```
./bin/jboss-eap-installation-manager.sh update perform --
dir eap-xp-5.0
```

Example

```
./bin/jboss-eap-installation-manager.sh update perform --
dir eap-xp-5.0
Updates found:
  org.wildfly.galleon-plugins:wildfly-galleon-plugins
6.3.1.Final-redhat-00001 ==>  6.3.2.Final-redhat-00001
  org.wildfly.wildfly-http-client:wildfly-http-
transaction-client    2.0.1.Final-redhat-00001 ==>
2.0.2.Final-redhat-00001
```

1.5. UPDATING JBOSS EAP XP INSTALLATION OFFLINE USING THE `JBOSS-EAP-INSTALLATION-MANAGER`

You can use the **`jboss-eap-installation-manager`** to update the JBoss EAP XP 5.0 installation offline.

Prerequisites

- You have installed a supported JDK.
- You have downloaded the **`jboss-eap-installation-manager`**. For more information about downloading **`jboss-eap-installation-manager`**, see the Installation Guide.
- You have downloaded or installed JBoss EAP XP 5.0 using one of the supported methods. For more information, see the Installation Guide.
- You have downloaded and extracted the latest offline repositories for JBoss EAP 8.0 and JBoss EAP XP 5.0.

Procedure

1. Stop the JBoss EAP server.
2. Open the terminal emulator and navigate to the **`jboss-eap-installation-manager`** directory.
3. Run this script from the **`jboss-eap-installation-manager`** directory to update the server components:

```
./bin/jboss-eap-installation-manager.sh update perform \
 --dir eap-xp-5.0  \
 --repositories
<JBOSS_EAP_XP_OFFLINE_REPO_PATH>,<FEATURE_PACK_OFFLINE_RE
PO>,<JBOSS_EAP_8.0_OFFLINE_REPO_PATH>
```

Additional resources

- For more information about how you can perform a two phase update operation offline see Updating feature packs on an offline JBoss EAP server.

1.6. REVERTING YOUR JBOSS EAP XP SERVER TO JBOSS EAP

You can use the **`jboss-eap-installation-manager`** to revert your JBoss EAP XP installation.

Prerequisites

- You have access to the internet.
- You have installed a supported JDK.
- You have downloaded the **`jboss-eap-installation-manager`**. For more information about downloading **`jboss-eap-installation-manager`**, see the installation guide.
- You have downloaded or installed JBoss EAP XP 5.0 using one of the supported methods. For more information, see the Installation Guide.

Procedure

1. Open the terminal emulator and navigate to the **`jboss-eap-installation-manager`** directory.
2. Run this script from the **`jboss-eap-installation-manager`** directory to investigate the history of all feature packs added to your JBoss EAP XP server:

```
./bin/jboss-eap-installation-manager.sh history --dir
eap-xp-5.0
```

3. Stop the JBoss EAP XP server.
4. Revert your to a version before JBoss EAP XP extension has been added:

```
./bin/jboss-eap-installation-manager.sh revert perform \
 --revision <REVISION_HASH> \
 --dir eap-xp-5.0
```

Additional resources

- For more information about how you can perform a two phase revert operation see Reverting installed feature packs

CHAPTER 2. UNDERSTAND MICROPROFILE

2.1. MICROPROFILE CONFIG

2.1.1. 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.

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.

MicroProfile Config functionality is implemented in JBoss EAP using the SmallRye Config component and is provided by the **`microprofile-config-smallrye`** subsystem.

Note

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

Important

If you are adding your own Config implementations, you need to use the methods in the latest version of the Config interface.

Additional Resources

- MicroProfile Config
- SmallRye Config
- Config implementations

2.1.2. MicroProfile Config sources supported in MicroProfile Config

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 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. MICROPROFILE FAULT TOLERANCE

2.2.1. About MicroProfile Fault Tolerance specification

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

The 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

- MicroProfile Fault Tolerance specification

2.2.2. MicroProfile Fault Tolerance in JBoss EAP

The **`microprofile-fault-tolerance-smallrye`** subsystem provides support for 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 the Jakarta Contexts and Dependency Injection 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 MicroProfile Fault Tolerance:

- **`io.smallrye.faulttolerance.mainThreadPoolSize`**
- **`io.smallrye.faulttolerance.mainThreadPoolQueueSize`**

Additional Resources

- MicroProfile Fault Tolerance Specification
- SmallRye Fault Tolerance project

2.3. MICROPROFILE HEALTH

2.3.1. 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.

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

The 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.

Startup

Determines whether an application has already started. The annotation **`@Startup`** provides this health check.

The **`@Health`** annotation was removed in MicroProfile Health 3.0.

MicroProfile Health 3.1 includes a new **`Startup`** health check probe.

For more information about the changes in MicroProfile Health 3.1, see Release Notes for MicroProfile Health 3.1.

Important

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

Additional Resources

- Global status when probes are not defined
- SmallRye Health
- MicroProfile Health
- Custom health check example

2.4. MICROPROFILE JWT

2.4.1. MicroProfile JWT integration in JBoss EAP

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

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

- Detecting deployments that use MicroProfile JWT security.
- Activating support for 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 MicroProfile JWT integration in JBoss EAP.

Additional Resources

- SmallRye JWT

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

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 MicroProfile JWT deployment.

As the MicroProfile JWT deployment is configured entirely within the 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. MicroProfile JWT activation in JBoss EAP

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

The 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 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 MicroProfile JWT in JBoss EAP

The MicroProfile JWT implementation in JBoss EAP has certain limitations.

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

- The 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. MICROPROFILE OPENAPI

2.5.1. MicroProfile OpenAPI in JBoss EAP

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

The 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.

2.6. MICROPROFILE TELEMETRY

2.6.1. MicroProfile Telemetry in JBoss EAP

MicroProfile Telemetry provides tracing functionality for applications based on OpenTelemetry. 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.

MicroProfile Telemetry expands on the OpenTelemetry subsystem and adds support for MicroProfile Config. This allows users to configure OpenTelemetry using MicroProfile Config.

> **Note**

There are no configurable resources or attributes in the MicroProfile Telemetry subsystem.

Additional resources

- Observability in JBoss EAP
- MicroProfile Telemetry subsystem configuration in WildFly Admin guide
- OpenTelemetry documentation
- @WithSpan annotations OpenTelemetry documentation
- Baggage API OpenTelemetry documentation

2.7. MICROPROFILE REST CLIENT

2.7.1. MicroProfile REST client

JBoss EAP XP 5.0.0 supports the MicroProfile REST client 2.0 that builds on Jakarta RESTful Web Services 2.1.6 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`**
- Jakarta Contexts and Dependency Injection integration
- Access to server-sent events (SSE)

2.7.2. The `resteasy.original.webapplicationexception.behavior` MicroProfile Config property

MicroProfile Config is the name of a specification that developers can use to configure applications and microservices to run in multiple environments without having to modify or repackage those apps. Previously, MicroProfile Config was available for JBoss EAP as a technology preview, but it has since been removed. MicroProfile Config is now available only

on JBoss EAP XP.

Defining the **`resteasy.original.webapplicationexception.behavior`** MicroProfile Config property

You can set the **`resteasy.original.webapplicationexception.behavior`** parameter as either a **`web.xml`** servlet property or a system property. Here's an example of one such servlet property in **`web.xml`**:

```
<context-param>
    <param-
name>resteasy.original.webapplicationexception.behavior</param-
name>
    <param-value>true</param-value>
</context-param>
```

You can also use MicroProfile Config to configure any other RESTEasy property.

Additional resources

- For more information about MicroProfile Config on JBoss EAP XP, see *Understand MicroProfile*.
- For more information about the MicroProfile REST Client, see MicroProfile REST Client.
- For more information about RESTEasy, see Jakarta RESTful Web Services Request Processing.

2.8. MICROPROFILE REACTIVE MESSAGING

2.8.1. MicroProfile Reactive Messaging

When you upgrade to JBoss EAP XP 5.0.0, you can enable the newest version of MicroProfile Reactive Messaging, which includes reactive messaging extensions and subsystems.

A "reactive stream" is a succession of event data, along with processing protocols and standards, that is pushed across an asynchronous boundary (like a scheduler) without any buffering. An "event" might be a scheduled and repeating temperature check in a weather app, for example. The primary benefit of reactive streams is the seamless interoperability of your various applications and implementations.

Reactive messaging provides a framework for building event-driven, data-streaming, and event-sourcing applications. Reactive messaging results in the constant and smooth exchange of event data, the reactive stream, from one app to another. You can use MicroProfile Reactive Messaging for asynchronous messaging through reactive streams so that your application can interact with others, like Apache Kafka, for example.

After you upgrade your instance of MicroProfile Reactive Messaging to the latest version, you can do the following:

- Provision a server with MicroProfile Reactive Messaging for the Apache Kafka data-streaming platform.

- Interact with reactive messaging in-memory and backed by Apache Kafka topics through the latest reactive messaging APIs.
- Use any metric system available to determine the number of messages streamed on a given channel.

Additional resources

- For more information about Apache Kafka, see What is Apache Kafka?

2.8.2. MicroProfile Reactive Messaging connectors

You can use connectors to integrate MicroProfile Reactive Messaging with a number of external messaging systems. MicroProfile for JBoss EAP comes with an Apache Kafka connector, and an Advanced Message Queuing Protocol (AMQP) connector. Use the Eclipse MicroProfile Config specification to configure your connectors.

MicroProfile Reactive Messaging connectors and incorporated layers

MicroProfile Reactive Messaging includes the following connectors:

- Kafka connector

  The **`microprofile-reactive-messaging-kafka`** layer incorporates the Kafka connector.

- AMQP connector

  The **`microprofile-reactive-messaging-amqp`** layer incorporates the AMQP connector.

Both the connector layers include the **`microprofile-reactive-messaging`** Galleon layer. The **`microprofile-reactive-messaging`** layer provides the core MicroProfile Reactive Messaging functionality.

Table 2.1. Reactive messaging and connector Galleon layers

Layer	Definition
`microprofile-reactive-streams-operators`	- Provides MicroProfile Reactive Streams Operators APIs and supporting implementing modules. - Contains MicroProfile Reactive Streams Operators with SmallRye extension and subsystem. - Depends on **`cdi`** layer. - **`cdi`** stands for Jakarta Contexts and Dependency Injection; provides subsystems that add **`@Inject`** functionality.

Layer	Definition
microprofile-reactive-messaging	• Provides MicroProfile Reactive Messaging APIs and supporting implementing modules. • Contains MicroProfile with SmallRye extension and subsystem. • Depends on **microprofile-config** and **microprofile-reactive-streams-operators** layers.
microprofile-reactive-messaging-kafka	• Provides Kafka connector modules that enable MicroProfile Reactive Messaging to interact with Kafka. • Depends on **microprofile-reactive-messaging** layer.
microprofile-reactive-messaging-amqp	• Provides AMQP connector modules that enable MicroProfile Reactive Messaging to interact with AMQP clients. • Depends on **microprofile-reactive-messaging** layer.

2.8.3. The Apache Kafka event streaming platform

Apache Kafka is an open source distributed event (data) streaming platform that can publish, subscribe to, store, and process streams of records in real time. It handles event streams from multiple sources and delivers them to multiple consumers, moving large amounts of data from points A to Z and everywhere else, all at the same time. MicroProfile Reactive Messaging uses Apache Kafka to deliver these event records in as few as two microseconds, store them safely in distributed, fault-tolerant clusters, all while making them available across any team-defined zones or geographic regions.

Additional resources

- What is Apache Kafka?
- Red Hat AMQ

CHAPTER 3. ADMINISTER

MICROPROFILE IN JBOSS EAP

3.1. MICROPROFILE TELEMETRY ADMINISTRATION

3.1.1. Add MicroProfile Telemetry subsystem using the management CLI

The MicroProfile Telemetry component is integrated into the default MicroProfile configuration through the **`microprofile-telemetry`** subsystem. You can also add the MicroProfile Telemetry subsystem using the management CLI if the subsystem is not included.

Prerequisites

- The OpenTelemetry subsystem must be added to the configuration before adding the MicroProfile Telemetry subsystem. The MicroProfile Telemetry subsystem depends on the OpenTelemetry subsystem.

Procedure

1. Open your terminal.
2. Run the following command:

```
$ <JBOSS_HOME>/bin/jboss-cli.sh -c <<EOF
    if (outcome != success) of /
subsystem=opentelemetry:read-resource
        /
extension=org.wildfly.extension.opentelemetry:add()
        /subsystem=opentelemetry:add()
    end-if
    /
extension=org.wildfly.extension.microprofile.telemetry:ad
d
    /subsystem=microprofile-telemetry:add
    reload
EOF
```

3.1.2. Enable MicroProfile Telemetry subsystem

MicroProfile Telemetry is disabled by default and must be enabled on a per-application basis.

Prerequisites

- The MicroProfile Telemetry subsystem has been added to the configuration.

- The OpenTelemetry subsystem has been added to the configuration.

Procedure

1. Open your **`microprofile-config.properties`** file.
2. Set the **`otel.sdk.disabled`** property to **`false`**:

```
otel.sdk.disabled=false
```

3.1.3. Override server configuration using MicroProfile Config

You can override server configuration for individual applications in the MicroProfile Telemetry subsystem using MicroProfile Config.

For example, the service name used in exported traces by default is the same as the deployment archive. If the deployment archive is set to **`my-application-1.0.war`**, the service name will be the same. To override this configuration, you can change the value of the **`otel.service.name`** property in your configuration file:

```
otel.service.name=My Application
```

3.2. 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. Configuring root directories as ConfigSources

You can define a directory as a root directory for multiple MicroProfile ConfigSource directories using the **`root`** attribute.

The nested **`root`** attribute is part of the **`dir`** complex attribute for the **`/subsystem=microprofile-config-smallrye/config-source=*`** resource. This eliminates the need to specify multiple ConfigSource directories if they share the same root directory.

Any files directly within the root directory are ignored. They will not be used for configuration. Top-level directories are treated as ConfigSources. Any nested directories will also be ignored.

Note

ConfigSources for top-level directories are assigned the **`ordinal`** of the **`/subsystem=microprofile-config-smallrye/config-source=*`** resource by default.

If the top-level directory contains a **`config_ordinal`** file, the value specified in the file will **override** the default **`ordinal`** value. If two top-level directories with the same **`ordinal`** contain the same entry, the names of the directories are sorted alphabetically and the first directory is used.

Prerequisites

- You have installed the MicroProfile Config extension and enabled the **`microprofile-config-smallrye`** subsystem.

Procedure

1. Open your terminal.
2. Create a directory where you want to store your files:

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

3. Navigate to the directory that you created:

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

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

```
touch name
```

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

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

6. Run the following command in the CLI to create a ConfigSource in which the filename is the property and the file contains the value of the property:

```
/subsystem=microprofile-config-smallrye/config-
source=prop-files:add(dir={path=/etc/config, root=true})
```

7. This results in the XML configuration:

```
<subsystem
```

```
xmlns="urn:wildfly:microprofile-config-smallrye:2.0">
 <config-source name="prop-files">
   <dir path="/etc/config" root="true"/>
 </config-source>
</subsystem>
```

3.2.4. 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.

3.2.5. 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.

3.3. 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

- JBoss EAP 8.0 with JBoss EAP XP 5.0 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. 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 MicroProfile Health

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

Procedure

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

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

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

```
reload
```

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

3.4.5. Readiness probes that determine server health and readiness

JBoss EAP XP 5.0.0 supports three readiness probes to determine server health and readiness.

- **`server-status`** – returns **`UP`** when the server-state is **`running`**.

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

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

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

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

Additional resources

3.4.6. Global status when probes are not defined

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

These attributes allow applications to report 'DOWN' until their probes verify that the application is ready, live, or started up. 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 **`:empty-startup-checks-status`** attribute specifies the global status for **`startup`** probes if no **`startup`** probes have been defined:

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

```
}
```

The **/health** HTTP endpoint and the **:check** operation that check **readiness**, **liveness**, and **startup** 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. MICROPROFILE JWT CONFIGURATION

3.5.1. Enabling microprofile-jwt-smallrye subsystem

The 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

- JBoss EAP 8.0 with JBoss EAP XP 5.0 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. MICROPROFILE OPENAPI ADMINISTRATION

3.6.1. Enabling 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 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.

3.6.2. Requesting an MicroProfile OpenAPI document using Accept HTTP header

Request an 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 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.

3.6.3. Requesting an MicroProfile OpenAPI document using an HTTP parameter

Request an 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 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.

3.6.4. 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 Jakarta RESTful Web Services 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.

3.6.5. 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()
```

3.7. MICROPROFILE REACTIVE MESSAGING ADMINISTRATION

3.7.1. Configuring the required MicroProfile reactive messaging extension and subsystem for JBoss EAP

If you want to enable asynchronous reactive messaging to your instance of JBoss EAP, you must add its extension through the JBoss EAP management CLI.

Prerequisites

- You added the Reactive Streams Operators with SmallRye extension and subsystem. For more information, see MicroProfile Reactive Streams Operators Subsystem Configuration: Required Extension.
- You added the Reactive Messaging with SmallRye extension and subsystem.

Procedure

1. Open the JBoss EAP management CLI.
2. Enter the following code:

```
[standalone@localhost:9990 /] /
extension=org.wildfly.extension.microprofile.reactive-
messaging-smallrye:add
{"outcome" => "success"}

[standalone@localhost:9990 /] /subsystem=microprofile-
reactive-messaging-smallrye:add
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

Note

If you provision a server using Galleon, either on OpenShift or not, make sure you include the **`microprofile-reactive-messaging`** Galleon layer to get the core MicroProfile 2.0.1 and reactive messaging functionality, and to enable the required subsystems and extensions. Note that this configuration does not contain the JBoss EAP modules you need to enable connectors. Use the **`microprofile-reactive-messaging-kafka`** layer or the **`microprofile-reactive-messaging-amqp`** layer to enable the Kafka connector or the AMQP connector, respectively.

Verification

You have successfully added the required MicroProfile Reactive Messaging extension and subsystem for JBoss EAP if you see **`success`** in two places in the resulting code in the management CLI.

Tip

If the resulting code says **`reload-required`**, you have to reload your server configuration to completely apply all of your changes. To reload, in a standalone server CLI, enter **`reload`**.

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 5.0.0 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 MicroProfile.	Includes information about the server, including subsystems, networking, deployments, socket bindings, and other configurable details.	Excludes the following capabilities: - Jakarta Enterprise Beans - Messaging - Jakarta EE Batch - Jakarta Server Faces - Jakarta Enterprise Beans 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 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		

Configuration File	Purpose	Included capabilities	Excluded capabilities
	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 MicroProfile configuration, use the **`-c`** argument. For example,

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

3.8.2. Updating standalone configurations with MicroProfile subsystems and extensions

You can update standard standalone server configuration files with 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 MicroProfile subsystems and extensions if they do not exist in the standalone configuration file:

- **`microprofile-config-smallrye`**
- **`microprofile-fault-tolerance-smallrye`**
- **`microprofile-health-smallrye`**
- **`microprofile-jwt-smallrye`**
- **`microprofile-openapi-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 8.0 with JBoss EAP XP 5.0 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 MicroProfile subsystems and extensions.

CHAPTER 4. DEVELOP MICROPROFILE APPLICATIONS FOR JBOSS EAP

To get started with developing applications that use MicroProfile APIs, create a Maven project and define the required dependencies. Use the JBoss EAP MicroProfile Bill of Materials (BOM) to control the versions of runtime Maven dependencies in the application Project Object Model (POM).

After you create a Maven project, refer to the JBoss EAP XP Quickstarts for information about developing applications for specific MicroProfile APIs. For more information, see JBoss EAP XP Quickstarts.

4.1. CREATING A MAVEN PROJECT WITH `MAVEN-ARCHETYPE-WEBAPP`

Use the **`maven-archetype-webapp`** archetype to create a Maven project for building applications for JBoss EAP deployment. Maven provides different archetypes for creating projects based on templates specific to project types. The **`maven-archetype-webapp`** creates a project with the structure required to develop simple web-applications.

Prerequisites

- You have installed Maven. For more information, see Downloading Apache Maven.

Procedure

1. Set up a Maven project by using the **`mvn`** command. The command creates the directory structure for the project and the **`pom.xml`** configuration file.

```
$ mvn archetype:generate                                \
```

```
-DgroupId=<group_id>                                \ 1
-DartifactId=<artifact_id>                          \ 2
-DarchetypeGroupId=org.apache.maven.archetypes \ 3
-DarchetypeArtifactId=maven-archetype-webapp   \ 4
-DinteractiveMode=false                             5
```

groupID uniquely identifies the project.

artifactID is the name for the generated **jar** archive.

archetypeGroupID is the unique ID for **maven-archetype-webapp**.

archetypeArtifactId is the artifact ID for **maven-archetype-webapp**.

InteractiveMode instructs Maven to use the supplied parameters rather than starting in interactive mode.

2. Navigate to the generated directory.

3. Open the generated **pom.xml** configuration file in a text editor.

4. Remove the content inside the **<project>** section of the **pom.xml** configuration file after the **<name>helloworld Maven Webapp</name>** line.

   Ensure that the file looks like this:

```
<?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>${group_id}</groupId>
    <artifactId>${artifact_id}</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>
    <name>${artifact_id} Maven Webapp</name>
```

```
</project>
```

The content was removed because it is not required for the application.

Next steps

- Defining properties in a Maven project.

4.2. DEFINING PROPERTIES IN A MAVEN PROJECT

You can define properties in a Maven **`pom.xml`** configuration file as place holders for values. Define the value for JBoss EAP XP server as a property to use the value consistently in the configuration.

Prerequisites

- You have initialized a Maven project.

  For more information, see Creating a Maven project with **`maven-archetype-webapp`**.

Procedure

- Define a property **`<version.server>`** as the JBoss EAP XP version on which you will deploy the configured application.

```
<project>
    ...
    <properties>
        <project.build.sourceEncoding>UTF-8</
project.build.sourceEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <version.server>5.0.0.GA</version.server>
    </properties>
</project>
```

Next steps

- Defining the repositories in a Maven project.

4.3. DEFINING THE REPOSITORIES IN A MAVEN PROJECT

Define the artifact and plug-in repositories in which Maven looks for artifacts and plug-ins to download.

Prerequisites

- You have initialized a Maven project.

For more information, see Creating a Maven project with **`maven-archetype-webapp`**.

Procedure

1. Define the artifacts repository.

```
<project>
    ...
    <repositories>
        <repository>
 1
            <id>jboss-public-maven-repository</id>
            <name>JBoss Public Maven Repository</name>
            <url>https://repository.jboss.org/nexus/
content/groups/public/</url>
            <releases>
                <enabled>true</enabled>
                <updatePolicy>never</updatePolicy>
            </releases>
            <snapshots>
                <enabled>true</enabled>
                <updatePolicy>never</updatePolicy>
            </snapshots>
            <layout>default</layout>
        </repository>
        <repository>
 2
            <id>redhat-ga-maven-repository</id>
            <name>Red Hat GA Maven Repository</name>
            <url>https://maven.repository.redhat.com/ga/</
url>
            <releases>
                <enabled>true</enabled>
                <updatePolicy>never</updatePolicy>
            </releases>
            <snapshots>
                <enabled>true</enabled>
                <updatePolicy>never</updatePolicy>
            </snapshots>
            <layout>default</layout>
        </repository>
    </repositories>
</project>
```

The Red Hat GA Maven repository provides all the productized JBoss EAP and other Red Hat artifacts.

The JBoss Public Maven Repository provides artifacts such as WildFly Maven plug-ins

2. Define the plug-ins repository.

```
<project>
    ...
    <pluginRepositories>
        <pluginRepository>
            <id>jboss-public-maven-repository</id>
            <name>JBoss Public Maven Repository</name>
            <url>https://repository.jboss.org/nexus/
content/groups/public/</url>
            <releases>
               <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </pluginRepository>
        <pluginRepository>
            <id>redhat-ga-maven-repository</id>
            <name>Red Hat GA Maven Repository</name>
            <url>https://maven.repository.redhat.com/ga/</
url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </pluginRepository>
    </pluginRepositories>
</project>
```

Next steps

- Importing the JBoss EAP MicroProfile BOM as dependency management in a Maven project.

4.4. IMPORTING THE JBOSS EAP MICROPROFILE BOM AS DEPENDENCY MANAGEMENT IN A MAVEN PROJECT

Import the JBoss EAP MicroProfile Bill of Materials (BOM) to control the versions of runtime Maven dependencies. When you specify a BOM in the **<dependencyManagement>** section, you do not need to individually specify the versions of the Maven dependencies defined in the **provided** scope.

Prerequisites

- You have initialized a Maven project.

  For more information, see Creating a Maven project with **maven-archetype-webapp**.

Procedure

1. Add a property for the BOM version in the properties section of the **pom.xml** configuration file.

```
<properties>
    ...
    <version.bom.microprofile>${version.server}</
version.bom.ee>
</properties>
```

   The value defined in the property **<version.server>** is used as the value for the BOM version.

2. Import the JBoss EAP BOMs dependency management.

```
<project>
    ...
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.jboss.bom</groupId>
 1
                <artifactId>jboss-eap-xp-microprofile</
artifactId>  2
                <version>${version.bom.microprofile}</
version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

---

groupID of the JBoss EAP-provided BOM.

---

artifactID of the JBoss EAP-provided BOM that provides supported JBoss EAP MicroProfile APIs.

Optionally, you can import the JBoss EAP EE with Tools Bill to your project. For more information, see Importing the JBoss EAP BOMs as dependency management in a Maven project.

Next steps

- Adding plug-in management in a Maven project

4.5. IMPORTING THE JBOSS EAP BOMS AS DEPENDENCY MANAGEMENT IN A MAVEN PROJECT

You can optionally import the JBoss EAP EE With Tools Bill of materials (BOM). The JBoss EAP BOM provides supported JBoss EAP Java EE APIs plus additional JBoss EAP API JARs and client BOMs. You only need to import this BOM if your application requires Jakarta EE APIs in addition to the Microprofile APIs.

Prerequisites

- You have initialized a Maven project.

  For more information, see Creating a Maven project with **`maven-archetype-webapp`**.

Procedure

1. Add a property for the BOM version in the properties section of the **`pom.xml`** configuration file.

```
<properties>
    ....
    <version.bom.ee>8.0.0.GA-redhat-00009</version.bom.ee>
</properties>
```

2. Import the JBoss EAP BOMs dependency management.

```
<project>
    ...
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.jboss.bom</groupId>
 1
                <artifactId>jboss-eap-ee-with-tools</
artifactId>  2
                <version>${version.bom.ee}</version>
                <type>pom</type>
                <scope>import</scope>
```

```
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

groupID of the JBoss EAP-provided BOM.

artifactID of the JBoss EAP-provided BOM that provides supported JBoss EAP Java EE APIs plus additional JBoss EAP API JARs and client BOMs, and development tools such as Arquillian.

Next steps

- Adding plug-in management in a Maven project

4.6. ADDING PLUG-IN MANAGEMENT IN A MAVEN PROJECT

Add Maven plug-in management section to the **`pom.xml`** configuration file to get plug-ins required for Maven CLI commands.

Prerequisites

- You have initialized a Maven project.

  For more information, see Creating a Maven project with **`maven-archetype-webapp`**.

Procedure

1. Define the versions for **`wildfly-maven-plugin`** and **`maven-war-plugin`**, in the **`<properties>`** section.

```
<properties>
    ...
    <version.plugin.wildfly>4.2.1.Final</
version.plugin.wildfly>
    <version.plugin.war>3.3.2</version.plugin.war>
</properties>
```

2. Add **`<pluginManagement>`** in **`<build>`** section inside the **`<project>`** section.

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

```
            <plugins>
                <plugin>     1
                    <groupId>org.wildfly.plugins</groupId>
                    <artifactId>wildfly-maven-plugin</
artifactId>
                    <version>${version.plugin.wildfly}</
version>
                </plugin>
                <plugin>
 2
                    <groupId>org.apache.maven.plugins</
groupId>
                    <artifactId>maven-war-plugin</
artifactId>
                    <version>${version.plugin.war}</
version>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
</project>
```

You can use the **`wildfly-maven-plugin`** to deploy an application to JBoss EAP using the **`wildfly:deploy`** command.

You need to manage the war plugin version to ensure compatibility with JDK17+.

Next steps

- Verifying a maven project

4.7. VERIFYING A MAVEN PROJECT

Verify that the Maven project you configured builds.

Prerequisites

- You have defined Maven properties.

  For more information, see Defining properties in a Maven project.

- You have defined Maven repositories.

For more information, see Defining the repositories in a Maven project.

- You have imported the JBoss EAP Bill of materials (BOMs) as dependency management.

  For more information, see Importing the JBoss EAP MicroProfile BOM as dependency management in a Maven project.

- You have added plug-in management.

  For more information, see Adding plugin management in Maven project for a server hello world application.

Procedure

- Install the Maven dependencies added in the **`pom.xml`** locally.

```
$ mvn package
```

You get an output similar to the following:

```
...
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
...
```

For more information about developing applications for specific MicroProfile APIs, see JBoss EAP XP Quickstarts.

Additional resources

- The bootable JAR

CHAPTER 5. UNDERSTAND MICROMETER INTEGRATION

5.1. MICROMETER IN JBOSS EAP

Micrometer integration in JBoss EAP introduces a vendor-neutral observability layer with a reusable API for registering and tracking performance metrics across applications. This extension integrates with Micrometer, allowing deployed

applications to access its API and display application-specific metrics alongside the server metrics provided by the extension.

Note

JBoss EAP uses the existing metrics subsystem. You must manually add and configure this extension.

Additional resources

- Micrometer Metrics subsystem configuration in WildFly Admin guide
- Micrometer documentation

CHAPTER 6. ADMINISTER MICROMETER IN JBOSS EAP

6.1. ADDING MICROMETER SUBSYSTEM USING THE MANAGEMENT CLI

The Micrometer subsystem enhances monitoring capabilities in JBoss EAP by facilitating comprehensive metrics gathering and publication. However, the **`org.jboss.extension.micrometer`** subsystem is available to all standalone configurations within the JBoss EAP distribution, but it must be added manually.

Prerequisites

- JBoss EAP 8.0 with JBoss EAP XP 5.0 is installed.
- You have access to the JBoss EAP management CLI and permissions to make configuration changes.

Procedure

1. Open your terminal.
2. Connect to the server by running the following command:

```
./jboss-cli.sh --connect
```

3. Check if the Micrometer extension is already added to the configuration by running the following command:

```
[standalone@localhost:9990 /] /
```

```
extension=org.wildfly.extension.micrometer:read-resource
```

4. If the Micrometer extension is not available, add it by running the following command:

```
[standalone@localhost:9990 /] /
extension=org.wildfly.extension.micrometer:add
```

5. Add the Micrometer subsystem with the required configuration. For example, specify the endpoint URL of the metrics collector by running the following command:

```
[standalone@localhost:9990 /] /
subsystem=micrometer:add(endpoint="http://localhost:4318/
v1/metrics")
```

6. Reload the server to apply the changes:

```
[standalone@localhost:9990 /] reload
```

Note

When the collector is not running or its collector endpoint is unavailable, then a warning message similar to the following is triggered:

```
11:28:16,581 WARNING
[io.micrometer.registry.otlp.OtlpMeterRegistry] (MSC
service thread 1-5) Failed to publish metrics to OTLP
receiver: java.net.ConnectException: Connection refused
```

By following these steps, you can add the Micrometer subsystem to your JBoss EAP server using the management CLI, enabling enhanced monitoring capabilities for your applications.

Additional resources

- Develop Micrometer application for JBoss EAP

CHAPTER 7. DEVELOP MICROMETER APPLICATION FOR JBOSS EAP

7.1. INTEGRATING MICROMETER METRICS IN JBOSS EAP

Using Micrometer, you can monitor and collect application metrics in JBoss EAP. Micrometer support provides the exposure of application metrics. The export process is PUSH-based, ensuring that metrics are sent to an OpenTelemetry Collector.

Prerequisites

- You have installed JDK 17.
- You have installed the Maven 3.6 or later version. For more information, see Downloading Apache Maven.
- You have installed Docker. For more information, see Get Docker.
- Optional: You have podman installed on your system. Use the latest podman version available on supported RHEL. For more information, see Red Hat JBoss Enterprise Application Platform 8.0 Supported Configurations.
- The **`configure-micrometer.cli`** file is available in the application root directory.

> **Note**
>
> The example in this section, including how to use the **`configure-micrometer.cli`** file, is based on the Micrometer Quickstart.

Procedure

1. Open a terminal.
2. Start JBoss EAP as a standalone server by using the following script:

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

> **Note**
>
> For Windows server, use the **`<EAP_HOME>\bin\standalone.bat`** script.

3. Open a new terminal.
4. Navigate to the application root directory.
5. Run the following command to configure the server:

```
$ <EAP_HOME>/bin/jboss-cli.sh --connect --file=configure-micrometer.cli
```

Note

For Windows server, use the ***<EAP_HOME>*\bin\jboss-cli.bat** script.

Replace *<EAP_HOME>* with the path to your server.

Expected output:

```
The batch executed successfully
process-state: reload-required
```

6. Reload the server with the following management command:

```
$ <EAP_HOME>/bin/jboss-cli.sh --connect --commands=reload
```

7. Create a configuration file named **docker-compose.yaml** with the following content:

```
version: "3"

services:
  otel-collector:
    image: otel/opentelemetry-collector
    command: [--config=/etc/otel-collector-config.yaml]
    volumes:
      - ./otel-collector-config.yaml:/etc/otel-collector-
config.yaml:Z
    ports:
      - 1888:1888 # pprof extension
      - 8888:8888 # Prometheus metrics exposed by the
collector
      - 8889:8889 # Prometheus exporter metrics
      - 13133:13133 # health_check extension
      - 4317:4317 # OTLP gRPC receiver
      - 4318:4318 # OTLP http receiver
      - 55679:55679 # zpages extension
      - 1234:1234 # /metrics endpoint
```

8. Create a configuration file named **otel-collector-config.yaml** with the following content:

```
extensions:
  health_check:
  pprof:
    endpoint: 0.0.0.0:1777
```

```
  zpages:
    endpoint: 0.0.0.0:55679

receivers:
  otlp:
    protocols:
      grpc:
      http:

processors:
  batch:

exporters:
  prometheus:
    endpoint: "0.0.0.0:1234"

service:
  pipelines:
    metrics:
      receivers: [otlp]
      processors: [batch]
      exporters: [prometheus]

  extensions: [health_check, pprof, zpages]
```

9. Start the collector server instance by running the following command:

```
$ docker-compose up
```

> **Note**
>
> You can also use Podman instead of Docker. If you choose Podman, then use the **`$ podman-compose up`** command instead of **`$ docker-compose up`**. If Docker or Podman is not supported in your environment, then see Otel Collector documentation for guidance on installing and running the OpenTelemetry Collector.

10. In the **`RootResource`** class, see how the **`MeterRegistry`** is injected into your class to ensure proper setup before registering the meters.

```
@Path("/")
@ApplicationScoped
public class RootResource {
    // ...
```

```
    @Inject
    private MeterRegistry registry;

    private Counter performCheckCounter;
    private Counter originalCounter;
    private Counter duplicatedCounter;

    @PostConstruct
    private void createMeters() {
        Gauge.builder("prime.highestSoFar", () ->
highestPrimeNumberSoFar)
                .description("Highest prime number so
far.")
                .register(registry);
        performCheckCounter = Counter
                .builder("prime.performedChecks")
                .description("How many prime checks have
been performed.")
                .register(registry);
        originalCounter = Counter
                .builder("prime.duplicatedCounter")
                .tags(List.of(Tag.of("type",
"original")))
                .register(registry);
        duplicatedCounter = Counter
                .builder("prime.duplicatedCounter")
                .tags(List.of(Tag.of("type", "copy")))
                .register(registry);
    }
    // ...
}
```

11. Inspect the **`checkIfPrime()`** method body to see how to use the registered meters within your application logic.

    For example:

```
@GET
@Path("/prime/{number}")
public String checkIfPrime(@PathParam("number") long
number) throws Exception {
    performCheckCounter.increment();

    Timer timer = registry.timer("prime.timer");

    return timer.recordCallable(() -> {
```

```
        if (number < 1) {
            return "Only natural numbers can be prime
numbers.";
        }

        if (number == 1) {
            return "1 is not prime.";
        }

        if (number == 2) {
            return "2 is prime.";
        }

        if (number % 2 == 0) {
            return number + " is not prime, it is
divisible by 2.";
        }

        for (int i = 3; i < Math.floor(Math.sqrt(number))
+ 1; i = i + 2) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                //
            }
            if (number % i == 0) {
                return number + " is not prime, is
divisible by " + i + ".";
            }
        }

        if (number > highestPrimeNumberSoFar) {
            highestPrimeNumberSoFar = number;
        }

        return number + " is prime.";
    });
}
```

12. Navigate to the application root directory.

Syntax

```
$ cd <path_to_application_root>/<application_root>
```

Example, in reference to the Micrometer Quickstart:

```
$ cd ~/quickstarts/micrometer
```

13. Compile and deploy the application with the following command:

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

This deploys **`micrometer/target/micrometer.war`** to the running server.

Verification

1. Access the application by using a web browser or you can run the following command.

```
$ curl http://localhost:8080/micrometer/prime/13
```

Expected output:

```
13 is prime.
```

CHAPTER 8. 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

Default **`cloud-default-mp-config`** layer provide a 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 5.0.0 for more information.

Additional resources

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

8.1. 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
   ```

8.2. BUILDING AND DEPLOYING JBOSS EAP XP APPLICATION IMAGES USING S2I

Follow the source-to-image (S2I) workflow to build reproducible container images for a JBoss EAP XP application. These generated container images include the application deployment and ready-to-run JBoss EAP XP servers.

The S2I workflow takes source code from a Git repository and injects it into a container that's based on the language and framework you want to use. After the S2I workflow is completed, the **`src`** code is compiled, the application is packaged and is deployed to the JBoss EAP XP server.

Prerequisites

- You have an active Red Hat customer account.
- You have a Registry Service Account. Follow the instructions on the Red Hat Customer Portal to create an authentication token using a registry service account.
- You have downloaded the OpenShift secret YAML file, which you can use to pull images from Red Hat Ecosystem Catalog. For more information, see OpenShift Secret.
- You used the **`oc login`** command to log in to OpenShift.
- You have installed Helm. For more information, see Installing Helm.
- You have installed the repository for the JBoss EAP Helm charts by entering this command in the management CLI:

```
$ helm repo add jboss-eap https://jbossas.github.io/eap-charts/
```

Procedure

1. Create a file named **`helm.yaml`** using the following YAML content:

```
build:
  uri: https://github.com/jboss-developer/jboss-eap-quickstarts.git
  ref: XP_5.0.0.GA
  contextDir: microprofile-config
  mode: s2i
deploy:
  replicas: 1
```

2. Use the following command to deploy your JBoss EAP XP application on Openshift.

```
$ helm install microprofile-config -f helm.yaml jboss-eap/eap-xp5
```

Note

This procedure is very similar to *Building application images using source-to-image in OpenShift*. For more information about that procedure see Using JBoss EAP on OpenShift Container Platform.

Verification

- Access the application using **`curl`**.

```
$ curl https://$(oc get route microprofile-config --
template='{{ .spec.host }}')/config/value
```

You get the output **`MyPropertyFileConfigValue`** confirming that the application is deployed.

8.3. 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-xp3-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 5.0.0 GA distribution, the Microprofile Config quickstart does not reply to HTTPS GET requests to the application's root context. This enhancement is only available in the {JBossXPShortName101} GA distribution.

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

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

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

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

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

```
$ oc scale deploymentconfig/eap-xp3-basic-app --replicas=3
```

Additional Resources

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

CHAPTER 9. CAPABILITY TRIMMING

When building a bootable JAR, you can decide which JBoss EAP features and subsystems to include.

Note

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

Additional resources

- About the bootable JAR

9.1. AVAILABLE JBOSS EAP LAYERS

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

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

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

Note

Only the listed layers are supported. Layers not listed here are not supported.

9.1.1. Base layers

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

datasources-web-server

This layer includes a servlet container and the ability to configure a datasource.

This layer does not include MicroProfile capabilities.

The following Jakarta EE specifications are supported in this layer:

- Jakarta JSON Processing 1.1
- Jakarta JSON Binding 1.0
- Jakarta Servlet 4.0
- Jakarta Expression Language 3.0
- Jakarta Server Pages 2.3
- Jakarta Standard Tag Library 1.2

- Jakarta Concurrency 1.1
- Jakarta Annotations 1.3
- Jakarta XML Binding 2.3
- Jakarta Debugging Support for Other Languages 1.0
- Jakarta Transactions 1.3
- Jakarta Connectors 1.7

jaxrs-server

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

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

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

The following MicroProfile capability is included in this layer:

- MicroProfile REST Client

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

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

cloud-server

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

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

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

- MicroProfile Health
- MicroProfile Config

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

- Jakarta Security 1.0

9.1.2. Decorator layers

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

ejb-lite

This decorator layer adds a minimal Jakarta Enterprise Beans implementation to the provisioned server. The following support is not included in this layer:

- IIOP integration
- MDB instance pool
- Remote connector resource

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

Jakarta Enterprise Beans

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

- MDB instance pool
- Remote connector resource

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

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

ejb-local-cache

This decorator layer adds local caching support for Jakarta Enterprise Beans to the provisioned server.

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

Note

This layer is not compatible with the **`ejb-dist-cache`** layer. If you include the **`ejb-dist-cache`** layer, you cannot include the **`ejb-local-cache`** layer. If you include both layers, the resulting build might include an unexpected Jakarta Enterprise Beans configuration.

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

ejb-dist-cache

This decorator layer adds distributed caching support for Jakarta Enterprise Beans to the provisioned server.

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

Note

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

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

jdr

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

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

Jakarta Persistence

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

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

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

Note

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

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

jpa-distributed

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

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

Note

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

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

Jakarta Server Faces

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

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

microprofile-platform

This decorator layer adds the following MicroProfile capabilities to the provisioned server:

- MicroProfile Config
- MicroProfile Fault Tolerance
- MicroProfile Health

- MicroProfile JWT
- MicroProfile OpenAPI

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

observability

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

- MicroProfile Health
- MicroProfile Config

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

remote-activemq

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

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

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

sso

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

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

web-console

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

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

web-clustering

This decorator layer adds support for distributable web applications by configuring a non-local Infinispan-based container web cache for data session handling suitable to clustering environments.

web-passivation

This decorator layer adds support for distributable web applications by configuring a local Infinispan-based container web cache for data session handling suitable to single node environments.

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

webservices

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

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

Additional resources

- Pooled Connection Factory Attributes

CHAPTER 10. ENABLE MICROPROFILE APPLICATION DEVELOPMENT FOR JBOSS EAP USING JBOSS TOOLS

If you want to incorporate MicroProfile capabilities in applications that you develop using JBoss Tools, you must enable MicroProfile support for JBoss EAP in JBoss Tools.

JBoss EAP expansion packs provide support for 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.

10.1. CONFIGURING JBOSS TOOLS TO USE MICROPROFILE CAPABILITIES

To enable MicroProfile support on JBoss EAP, register a new runtime server for JBoss EAP XP, and then create the new JBoss EAP 8.0 server.

Give the server an appropriate name that helps you recognize that it supports 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.

Note

If you set the **Target runtime** to **8.0** or a later runtime version in JBoss Tools, your project is compatible with the Jakarta EE 8 specification.

Prerequisites

- JBoss EAP 8.0 with JBoss EAP XP 5.0 has been installed.

Procedure

1. Set up the new server on the **`New Server`** dialog box.
    a. In the **Select server type** list, select *Red Hat JBoss Enterprise Application Platform 8.0*.
    b. In the **Server's host name** field, enter *localhost*.
    c. In the **Server name** field, enter *JBoss EAP 8.0 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-7.4*.

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 MicroProfile capabilities, or to begin using the MicroProfile quickstarts for JBoss EAP.

10.2. USING MICROPROFILE QUICKSTARTS FOR JBOSS TOOLS

Enabling the MicroProfile quickstarts makes the simple examples available to run and test on your installed server.

These examples illustrate the following MicroProfile capabilities.

- MicroProfile Config
- MicroProfile Fault Tolerance
- MicroProfile Health
- MicroProfile JWT
- MicroProfile OpenAPI
- MicroProfile REST Client

Procedure

1. Import the **`pom.xml`** file from the Quickstart Parent Artifact.
2. If you are using a quickstart that requires environment variables, configure those variables on the launch configuration on the server **Overview** dialog box.

   For example, the **`opentelemetry-tracing`** quickstart uses the following environment variable:

   - **`OTEL_COLLECTOR_HOST`**

Additional resources

About Microprofile

About JBoss Enterprise Application Platform expansion pack

Red Hat JBoss Enterprise Application Platform expansion pack Support and Life Cycle Policies

CHAPTER 11. THE BOOTABLE JAR

You can build and package a microservices application as a bootable JAR with the JBoss EAP JAR Maven plug-in. You can then run the application on a JBoss EAP bare-metal platform or a JBoss EAP OpenShift platform.

11.1. ABOUT THE BOOTABLE JAR

You can build and package a microservices application as a bootable JAR with the JBoss EAP JAR Maven plug-in.

A bootable JAR contains a server, a packaged application, and the runtime required to launch the server.

The JBoss EAP JAR Maven plug-in uses Galleon trimming capability to reduce the size and memory footprint of the server. Thus, you can configure the server according to your requirements, including only the Galleon layers that provide the capabilities that you need.

The JBoss EAP JAR Maven plug-in supports the execution of JBoss EAP CLI script files to customize your server configuration. A CLI script includes a list of CLI commands for configuring the server.

A bootable JAR is like a standard JBoss EAP server in the following ways:

- It supports JBoss EAP common management CLI commands.
- It can be managed using the JBoss EAP management console.

The following limitations exist when packaging a server in a bootable JAR:

- CLI management operations that require a server restart are not supported.
- The server cannot be restarted in admin-only mode, which is a mode that starts services related to server administration.
- If you shut down the server, updates that you applied to the server are lost.

Additionally, you can provision a hollow bootable JAR. This JAR contains only the server, so you can reuse the server to run a different application.

Additional resources

For information about capability trimming, see Capability Trimming.

11.2. JBOSS EAP JAR MAVEN PLUG-IN

You can use the JBoss EAP JAR Maven plug-in to build an application as a bootable JAR.

Check that you have the latest Maven plug-in such as **`9.minor.micro.Final-redhat-XXXXX`**, where *9* is the major

version, *minor* is the minor version, *micro* micro version and *X* is the Red Hat build number. For example: **`9.0.1.Final-redhat-00009`**.

In a Maven project, the **`src`** directory contains all the source files required to build your application. After the JBoss EAP JAR Maven plug-in builds the bootable JAR, the generated JAR is located in **`target/<application>-bootable.jar`**.

The JBoss EAP JAR Maven plug-in also provides the following functionality:

- Allows you to provision JBoss EAP XP server using JBoss EAP channels.
- Applies CLI script commands to the server.
- Uses the **`org.jboss.eap.xp:wildfly-galleon-pack`** Galleon feature pack and some of its layers for customizing the server configuration file.
- Supports the addition of extra files into the packaged bootable JAR, such as a keystore file.
- Includes the capability to create a hollow bootable JAR; that is, a bootable JAR that does not contain an application.

After you use the JBoss EAP JAR Maven plug-in to create the bootable JAR, you can start the application by issuing the following command. Replace **`target/myapp-bootable.jar`** with the path to your bootable JAR. For example:

```
$ java -jar target/myapp-bootable.jar
```

Note

To get a list of supported bootable JAR startup commands, append **`--help`** to the end of the startup command. For example, **`java -jar target/myapp-bootable.jar --help`**.

Additional resources

- For information about supported JBoss EAP Galleon layers, see Available JBoss EAP layers.
- For information about supported Galleon plug-ins to build feature packs for your project, see the WildFly Galleon Maven Plugin Documentation.
- For information about selecting methods to configure the JBoss EAP Maven repository, see Use the Maven Repository.
- For information about Maven project directories, see Introduction to the Standard Directory Layout in the *Apache Maven* documentation.

11.3. BOOTABLE JAR ARGUMENTS

View the arguments in the following table to learn about supported arguments for use with the bootable JAR.

Table 11.1. Supported bootable JAR executable arguments

Argument	Description
`--help`	Displays the help message for the specified command and exit.
`--cli-script=<path>`	Specifies the path to a JBoss CLI script that executes when starting the bootable JAR. If the path specified is relative, the path is resolved against the working directory of the Java VM instance used to launch the bootable JAR.
`--deployment=<path>`	Argument specific to the hollow bootable JAR. Specifies the path to the WAR, JAR, EAR file or exploded directory that contains the application you want to deploy on a server.
`--display-galleon-config`	Print the content of the generated Galleon configuration file.
`--install-dir=<path>`	By default, the JVM settings are used to create a *TEMP* directory after the bootable JAR is started. You can use the **`--install-dir`** argument to specify a directory to install the server.
`-secmgr`	Runs the server with a security manager installed.
`-b<interface>=<value>`	Set system property **`jboss.bind.address.<interface>`** to the given value. For example, **`bmanagement=IP_ADDRESS`**.

Argument	Description
`-b=<value>`	Set system property **`jboss.bind.address`**, which is used in configuring the bind address for the public interface. This defaults to 127.0.0.1 if no value is specified.
`-D<name>[=<value>]`	Specifies system properties that are set by the server at server runtime. The bootable JAR JVM does not set these system properties.
`--properties=<url>`	Loads system properties from a specified URL.
`-S<name>[=value]`	Set a security property.
`-u=<value>`	Set system property **`jboss.default.multicast.address`**, which is used in configuring the multicast address in the socket-binding elements in the configuration files. This defaults to 230.0.0.4 if no value is specified.
`--version`	Display the application server version and exit.

11.4. SPECIFYING GALLEON LAYERS FOR YOUR BOOTABLE JAR SERVER

You can specify Galleon layers to build a custom configuration for your server. Additionally, you can specify Galleon layers that you want excluded from the server.

Starting with JBoss EAP XP 5.0, it is necessary to configure the JBoss EAP JAR Maven plug-in with the JBoss EAP 8.0 and JBoss EAP XP 5.0 channels to retrieve the server artifacts. For more information about JBoss EAP Channels, see

managing JBoss EAP installation channels.

To specify the JBoss EAP and JBoss EAP XP channels for provisioning the latest JBoss EAP XP 5.0 server, follow this example:

Note

Use the **`<feature-pack-location>`** element to specify feature pack location. In the Maven plug-in configuration file, the following example specifies **`org.jboss.eap.xp:wildfly-galleon-pack`** within the **`<feature-pack-location>`** element.

```
<configuration>
        <channels>
                <channel>
                        <manifest>

<groupId>org.jboss.eap.channels</groupId>
                                <artifactId>eap-8.0</
artifactId>
                        </manifest>
                </channel>
                <channel>
                        <manifest>

<groupId>org.jboss.eap.channels</groupId>
                                <artifactId>eap-xp-5.0</
artifactId>
                        </manifest>
                </channel>
        </channels>
        <feature-pack-location>org.jboss.eap.xp:wildfly-
galleon-pack</feature-pack-location>
</configuration>
```

If you need to reference more than one feature pack, list them in the **`<feature-packs>`** element. The following example shows the addition of the JBoss EAP datasources feature pack to the **`<feature-packs>`** element:

```
<configuration>
        <feature-packs>
                <feature-pack>
                        <location>org.jboss.eap.xp:wildfly-
galleon-pack</location>
                </feature-pack>
                <feature-pack>
```

```
                        <location>org.jboss.eap:eap-
datasources-galleon-pack</location>
                </feature-pack>
        </feature-packs>
</configuration>
```

You can combine Galleon layers from multiple feature packs to configure the bootable JAR server to include only the supported Galleon layers that provide the capabilities that you need.

> **Note**
>
> On a bare-metal platform, if you do not specify Galleon layers in your configuration file then the provisioned server contains a configuration identical to that of a default **`standalone-microprofile.xml`** configuration.
>
> On an OpenShift platform, after you have added the **`<cloud/>`** configuration element in the plug-in configuration and you choose not to specify Galleon layers in your configuration file, the provisioned server contains a configuration that is adjusted for the cloud environment and is similar to a default **`standalone-microprofile-ha.xml`**.

Prerequisites

- Maven is installed.
- You have checked the latest Maven plug-in version, such as **`9.minor.micro.Final-redhat-XXXXX`**, where *9* is the major version, *minor* is the minor version, *micro* micro version and *X* is the Red Hat build number. For example: **`9.0.1.Final-redhat-00009`**.

> **Note**
>
> The examples shown in the procedure specify the following properties:
>
> - **`${bootable.jar.maven.plugin.version}`** for the Maven plug-in version.
>
> You must set these properties in your project. For example:
>
> ```
> <properties>
>     <bootable.jar.maven.plugin.version>9.0.1.Final-
> redhat-00009</bootable.jar.maven.plugin.version>
> </properties>
> ```

Procedure

1. Identify the supported JBoss EAP Galleon layers that provide the capabilities that you need to run your application.
2. Reference a JBoss EAP feature pack location in the **`<plugin>`** element of the Maven project **`pom.xml`** file. The

following example displays the inclusion of a single feature-pack, which includes the **`jaxrs-server`** base layer and the **`jpa-distributed`** layer . The **`jaxrs-server`** base layer provides additional support for the server.

```
<plugins>
        <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-jar-maven-plugin</
artifactId>

<version>${bootable.jar.maven.plugin.version}</version>
                <configuration>
                        <channels>
                                <channel>
                                        <manifest>

<groupId>org.jboss.eap.channels</groupId>

<artifactId>eap-8.0</artifactId>
                                        </manifest>
                                </channel>
                                <channel>
                                        <manifest>

<groupId>org.jboss.eap.channels</groupId>

<artifactId>eap-xp-5.0</artifactId>
                                        </manifest>
                                </channel>
                        </channels>
                        <feature-pack-
location>org.jboss.eap.xp:wildfly-galleon-pack</feature-
pack-location>
                        <layers>
                                <layer>jaxrs-server</layer>
                                <layer>jpa-distributed</
layer>
                        </layers>
                        <excluded-layers>
                                <layer>jpa</layer>
                        </excluded-layers>
                ...

                </plugins>
```

This example also shows the exclusion of the **jpa** layer from the project.

Note

If you include the **jpa-distributed** layer in your project, you must exclude the **jpa** layer from the **jaxrs-server** layer. The **jpa** layer configures a local infinispan hibernate cache, while the **jpa-distributed** layer configures a remote infinispan hibernate cache.

Additional resources

- For information about available base layers, see Base layers.
- For information about supported Galleon plug-ins to build feature packs for your project, see the WildFly Galleon Maven Plugin Documentation.
- For information about selecting methods to configure the JBoss EAP Maven repository, see Maven and the JBoss EAP MicroProfile Maven repository.
- For information about managing your Maven dependencies, see Dependency Management in the *Apache Maven Project* documentation.

11.5. USING A BOOTABLE JAR ON A JBOSS EAP BARE-METAL PLATFORM

You can package an application as a bootable JAR on a JBoss EAP bare-metal platform.

A bootable JAR contains a server, a packaged application, and the runtime required to launch the server.

This procedure demonstrates packaging the MicroProfile Config microservices application as a bootable JAR with the JBoss EAP JAR Maven plug-in. See MicroProfile Config development.

You can use CLI scripts to configure the server during the packaging of the bootable JAR.

Important

On building a web application that must be packaged inside a bootable JAR, you must specify **war** in the **<packaging>** element of your **pom.xml** file. For example:

```
<packaging>war</packaging>
```

This value is required to package the build application as a WAR file and not as the default JAR file.

In a Maven project that is used solely to build a hollow bootable JAR, set the packaging value to **pom**. For example:

```
<packaging>pom</packaging>
```

You are not limited to using **pom** packaging when you build a hollow bootable JAR for a Maven project. You can create one by specifying **true** in the **<hollow-jar>** element for any type of packaging, such as **war**. See Creating a hollow bootable JAR on a JBoss EAP bare-metal platform.

Prerequisites

- You have checked the latest Maven plug-in version, such as **9.minor.micro.Final-redhat-XXXXX**, where *9* is the major version, *minor* is the minor version, *micro* micro version and *X* is the Red Hat build number. For example: **9.0.1.Final-redhat-00009**.
- You have created a Maven project, and added dependencies for creating an MicroProfile application. See MicroProfile Config development.

Note

The examples shown in the procedure specify the following properties:

- **${bootable.jar.maven.plugin.version}** for the Maven plug-in version.

You must set these properties in your project. For example:

```
<properties>
    <bootable.jar.maven.plugin.version>9.0.1.Final-
redhat-00009</bootable.jar.maven.plugin.version>
</properties>
```

Procedure

1. Add the following content to the **<build>** element of the **pom.xml** file. For example:

```
<plugins>
        <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-jar-maven-plugin</
artifactId>

<version>${bootable.jar.maven.plugin.version}</version>
                <configuration>
                        <channels>
                                <channel>
                                        <manifest>
```

```
<groupId>org.jboss.eap.channels</groupId>

<artifactId>eap-8.0</artifactId>
                                        </manifest>
                                    </channel>
                                    <channel>
                                        <manifest>

<groupId>org.jboss.eap.channels</groupId>

<artifactId>eap-xp-5.0</artifactId>
                                        </manifest>
                                    </channel>
                            </channels>
                            <feature-pack-
location>org.jboss.eap.xp:wildfly-galleon-pack</feature-
pack-location>
                            <layers>
                                    <layer>jaxrs-server</layer>
                                    <layer>microprofile-
platform</layer>
                            </layers>
                    </configuration>
                    <executions>
                            <execution>
                                    <goals>
                                        <goal>package</
goal>
                                    </goals>
                            </execution>
                    </executions>
            </plugin>
</plugins>
```

Note

If you do not specify Galleon layers in your **`pom.xml`** file then the bootable JAR server contains a configuration that is identical to a **`standalone-microprofile.xml`** configuration.

2. Package the application as a bootable JAR:

```
$ mvn package
```

3. Start the application:

```
$ NAME="foo" java -jar target/microprofile-config-bootable.jar
```

Note

The example uses **`NAME`** as the environment variable, but you can choose to use **`jim`**, which is the default value.

Note

To view a list of supported bootable JAR arguments, append **`--help`** to the end of the **`java -jar target/microprofile-config-bootable.jar`** command.

4. Specify the following URL in your web browser to access the MicroProfile Config application:

```
http://localhost:8080/config/json
```

5. *Verification:* Test the application behaves properly by issuing the following command in your terminal:

```
curl http://localhost:8080/config/json
```

The following is the expected output:

```
{"result":"Hello foo"}
```

Additional resources

- For information about available MicroProfile Config functionality, see MicroProfile Config.
- For information about **`ConfigSources`**, see MicroProfile Config reference.

11.6. CREATING A HOLLOW BOOTABLE JAR ON A JBOSS EAP BARE-METAL PLATFORM

You can package an application as a hollow bootable JAR on a JBoss EAP bare-metal platform.

A hollow bootable JAR contains only the JBoss EAP server. The hollow bootable JAR is packaged by the JBoss EAP JAR Maven plug-in. The application is provided at server runtime. The hollow bootable JAR is useful if you need to re-use the server configuration for a different application.

Prerequisites

- You have created a Maven project, and added dependencies for creating an application. See MicroProfile Config development.
- You have completed the **`pom.xml`** file configuration steps outlined in Using a bootable JAR on a JBoss EAP bare-metal platform.
- You have checked the latest Maven plug-in version, such as **`9.minor.micro.Final-redhat-XXXXX`**, where *9* is the major version, *minor* is the minor version, *micro* micro version and *X* is the Red Hat build number. For example: **`9.0.1.Final-redhat-00009`**.

Procedure

1. To build a hollow bootable JAR, you must set the **`<hollow-jar>`** plug-in configuration element to true in the project **`pom.xml`** file. For example:

```
<plugins>
        <plugin>
            ...
            <configuration>
                <!-- This example configuration does not show a
complete plug-in configuration -->
                 ...
                <feature-pack-
location>org.jboss.eap.xp:wildfly-galleon-pack</feature-pack-
location>
                 <hollow-jar>true</hollow-jar>
            </configuration>
         </plugin>
</plugins>
```

Note

By specifying **`true`** in the **`<hollow-jar>`** element, the JBoss EAP JAR Maven plug-in does not include an application in the JAR.

1. Build the hollow bootable JAR:

```
$ mvn clean package
```

2. Run the hollow bootable JAR:

```
$ java -jar target/microprofile-config-bootable.jar --
```

```
deployment=target/microprofile-config.war
```

Important

To specify the path to the WAR file that you want to deploy on the server, use the following argument, where **<PATH_NAME>** is the path to your deployment.

```
--deployment=<PATH_NAME>
```

3. Access the application:

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

Note

To register your web application in the root directory, name the application **ROOT.war**.

Additional resources

- For information about available MicroProfile functionality, see MicroProfile Config.
- For more information about the JBoss EAP JAR Maven plug-in supported in JBoss EAP XP 5.0.0, see JBoss EAP Maven plug-in.

11.7. CLI SCRIPTS EXECUTED AT BUILD TIME

You can create CLI scripts to configure the server during the packaging of the bootable JAR.

A CLI script is a text file that contains a sequence of CLI commands that you can use to apply additional server configurations. For example, you can create a script to add a new logger to the **logging** subsystem.

You can also specify more complex operations in a CLI script. For example, you can group security management operations into a single command to enable HTTP authentication for the management HTTP endpoint.

Note

You must define CLI scripts in the **<cli-session>** element of the plug-in configuration before you package an application as a bootable JAR. This ensures the server configuration settings persist after packaging the bootable JAR.

Although you can combine predefined Galleon layers to configure a server that deploys your application, limitations do exist. For example, you cannot enable the HTTPS **`undertow`** listener using Galleon layers when packaging the bootable JAR. Instead, you must use a CLI script.

You must define the CLI scripts in the **`<cli-session>`** element of the **`pom.xml`** file. The following table shows types of CLI session attributes:

Table 11.2. CLI script attributes

Argument	Description
`script-files`	List of paths to script files.
`properties-file`	Optional attribute that specifies a path to a properties file. This file lists Java properties that scripts can reference by using the **`${my.prop}`** syntax. The following example sets **`public inet-address`** to the value of **`all.addresses`**: **`/interface=public:write-attribute(name=inet-address,value=${all.addresses})`**
`resolve-expressions`	Optional attribute that contains a boolean value. Indicates if system properties or expressions are resolved before sending the operation requests to the server. Value is **`true`** by default.

Note

- CLI scripts are started in the order that they are defined in the **`<cli-session>`** element of the **`pom.xml`** file.
- The JBoss EAP JAR Maven plug-in starts the embedded server for each CLI session. Thus, your CLI script does not have to start or stop the embedded server.

11.8. EXECUTING CLI SCRIPT AT RUNTIME

You can apply changes to the server configuration during runtime; this gives you the flexibility to adjust the server with respect to the execution context. However, the preferred way to apply changes to the server is during build time.

Procedure

- Launch the bootable JAR, and the **`--cli-script`** argument.

  For Example:

```
java -jar myapp-bootable.jar --cli-scipt=my-scli-scipt.cli
```

Note

- The CLI script must be a text file (UTF-8), the file extension if present is meaningless although **`.cli`** extension is advised.
- Operations that require your server to restart will terminate your bootable JAR instance.
- CLI commands such as **`connect`**, **`reload`**, **`shutdown`**, and any command related to embedded server are not operational.
- CLI commands such as **`jdbc-driver-info`** that cannot be executed in admin-mode are not supported.

Important

If you restart the server without executing the CLI script, your new server instance will not contain the changes from your previous server instance.

11.9. USING A BOOTABLE JAR ON A JBOSS EAP OPENSHIFT PLATFORM

11.9.1. Using oc command to do binary build

After you packaged an application as a bootable JAR, you can run the application on a JBoss EAP OpenShift platform.

Important

On OpenShift, you cannot use the EAP Operator automated transaction recovery feature with your bootable

JAR.

Prerequisites

- You have created a Maven project for MicroProfile Config development.
- You have checked the latest Maven plug-in version, such as **`9.minor.micro.Final-redhat-XXXXX`**, where *9* is the major version, *minor* is the minor version, *micro* micro version and *X* is the Red Hat build number. For example: **`9.0.1.Final-redhat-00009`**.

Note

The examples shown in the procedure specify the following properties:

- **`${bootable.jar.maven.plugin.version}`** for the Maven plug-in version.

You must set these properties in your project. For example:

```
<properties>
    <bootable.jar.maven.plugin.version>9.0.1.Final-
redhat-00009</bootable.jar.maven.plugin.version>
</properties>
```

Procedure

1. Add the following content to the **`<build>`** element of the **`pom.xml`** file. For example:

```
<plugins>
        <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-jar-maven-plugin</
artifactId>

<version>${bootable.jar.maven.plugin.version}</version>
                <configuration>
                        <channels>
                                <channel>
                                        <manifest>

<groupId>org.jboss.eap.channels</groupId>

<artifactId>eap-8.0</artifactId>
                                        </manifest>
                                </channel>
                                <channel>
```

```
                                        <manifest>
<groupId>org.jboss.eap.channels</groupId>
<artifactId>eap-xp-5.0</artifactId>
                                        </manifest>
                                </channel>
                        </channels>
                        <feature-pack-
location>org.jboss.eap.xp:wildfly-galleon-pack</feature-
pack-location>
                        <layers>
                                <layer>jaxrs-server</layer>
                                <layer>microprofile-
platform</layer>
                        </layers>
                        <cloud/>
                </configuration>
                <executions>
                        <execution>
                                <goals>
                                        <goal>package</
goal>
                                </goals>
                        </execution>
                </executions>
        </plugin>
</plugins>
```

Note

You must include the **<cloud/>** element in the **<configuration>** element of the plug-in configuration, so the JBoss EAP Maven JAR plug-in can identify that you choose the OpenShift platform.

2. Package the application:

```
$ mvn package
```

3. Log in to your OpenShift instance using the **oc login** command.

4. Create a new project in OpenShift. For example:

```
$ oc new-project bootable-jar-project
```

5. Enter the following **oc** commands to create an application image:

```
$ mkdir target/openshift && cp target/microprofile-
config-bootable.jar target/openshift   1

$ oc import-image ubi8/openjdk-17 --
from=registry.redhat.io/ubi8/openjdk-17 --confirm  2

$ oc new-build --strategy source --binary --image-stream
openjdk-17 --name microprofile-config-app  3

$ oc start-build microprofile-config-app --from-dir
target/openshift  4
```

Creates an openshift sub-directory in the target directory. The packaged application is copied into the created sub-directory.

Imports the latest OpenJDK 17 imagestream tag and image information into the OpenShift project.

Creates a build configuration based on the microprofile-config-app directory and the OpenJDK 17 imagestream.

Uses the **`target/openshift`** sub-directory as the binary input to build the application.

Note

OpenShift applies a set of CLI script commands to the bootable JAR configuration file to adjust it to the cloud environment. You can access this script by opening the **`bootable-jar-build-artifacts/generated-cli-script.txt`** file in the Maven project **`/target directory`**.

6. *Verification:*

View a list of OpenShift pods available and check the pods build statuses by issuing the following command:

```
$ oc get pods
```

Verify the built application image:

```
$ oc get is microprofile-config-app
```

The output shows the built application image details, such as name and image repository, tag, and so on. For the example in this procedure, the imagestream name and tag output displays **`microprofile-config-app:latest`**.

7. Deploy the application:

```
$ oc new-app microprofile-config-app

$ oc expose svc/microprofile-config-app
```

Important

To provide system properties to the bootable JAR, you must use the **`JAVA_OPTS_APPEND`** environment variable. The following example demonstrates usage of the **`JAVA_OPTS_APPEND`** environment variable:

```
$ oc new-app <_IMAGESTREAM_> -e JAVA_OPTS_APPEND="-
Xlog:gc*:file=/tmp/gc.log:time -
Dwildfly.statistics-enabled=true"
```

A new application is created and started. The application configuration is exposed as a new service.

8. *Verification*: Test the application behaves properly by issuing the following command in your terminal:

```
$ curl http://$(oc get route microprofile-config-app --
template='{{ .spec.host }}')/config/json
```

Expected output:

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

Additional resources

- For information about MicroProfile, see MicroProfile Config.
- For information about **`ConfigSources`**, see Default MicroProfile Config attributes.

11.10. CONFIGURE THE BOOTABLE JAR FOR OPENSHIFT

Before using your bootable JAR, you can configure JVM settings to ensure that your standalone server operates correctly

on JBoss EAP for OpenShift.

Use the **JAVA_OPTS_APPEND** environment variable to configure JVM settings. Use the **JAVA_ARGS** command to provide arguments to the bootable JAR.

You can use environment variables to set values for properties. For example, you can use the **JAVA_OPTS_APPEND** environment variable to set the **-Dwildfly.statistics-enabled** property to **true**:

```
JAVA_OPTS_APPEND="-Xlog:gc*:file=/tmp/gc.log:time -
Dwildfly.statistics-enabled=true"
```

Statistics are now enabled for your server.

> **Note**
>
> Use the **JAVA_ARGS** environment variable, if you need to provide arguments to the bootable JAR.

JBoss EAP for OpenShift provides a JDK 17 image. To run the application associated with your bootable JAR, you must first import the latest OpenJDK 17 imagestream tag and image information into your OpenShift project. You can then use environment variables to configure the JVM in the imported image.

You can apply the same configuration options for configuring the JVM used for JBoss EAP for OpenShift S2I image, but with the following differences:

- Optional: The **-Xlog** capability is not available, but you can set garbage collection logging by enabling **-Xlog:gc**. For example: **JAVA_OPTS_APPEND="-Xlog:gc*:file=/tmp/gc.log:time"**.
- To increase initial metaspace size, you can set the **GC_METASPACE_SIZE** environment variable. For best metadata capacity performance, set the value to **96**.
- For better random file generation, use the **JAVA_OPTS_APPEND** environment variable to set **java.security.egd** property as **-Djava.security.egd=file:/dev/urandom**.

These configurations improve the memory settings and garbage collection capability of JVM when running on your imported OpenJDK 17 image.

11.11. USING A CONFIGMAP IN YOUR APPLICATION ON OPENSHIFT

For OpenShift, you can use a deployment controller (dc) to mount the configmap into the pods used to run the application.

A **ConfigMap** is an OpenShift resource that is used to store non-confidential data in key-value pairs.

After you specify the **microprofile-platform** Galleon layer to add **microprofile-config-smallrye** subsystem

and any extensions to the server configuration file, you can use a CLI script to add a new **`ConfigSource`** to the server configuration. You can save CLI scripts in an accessible directory, such as the **`/scripts`** directory, in the root directory of your Maven project.

MicroProfile Config functionality is implemented in JBoss EAP using the SmallRye Config component and is provided by the **`microprofile-config-smallrye`** subsystem. This subsystem is included in the **`microprofile-platform`** Galleon layer.

Prerequisites

- You have installed Maven.
- You have configured the JBoss EAP Maven repository.
- You have packaged an application as a bootable JAR and you can run the application on a JBoss EAP OpenShift platform. For information about building an application as a bootable JAR on an OpenShift platform, see Using a bootable JAR on a JBoss EAP OpenShift platform.

Procedure

1. Create a directory named **`scripts`** at the root directory of your project. For example:

```
$ mkdir scripts
```

2. Create a **`cli.properties`** file and save the file in the **`/scripts`** directory. Define the **`config.path`** and the **`config.ordinal`** system properties in this file. For example:

```
config.path=/etc/config
config.ordinal=200
```

3. Create a CLI script, such as **`mp-config.cli`**, and save it in an accessible directory in the bootable JAR, such as the **`/scripts`** directory. The following example shows the contents of the **`mp-config.cli`** script:

```
# config map

/subsystem=microprofile-config-smallrye/config-source=os-
map:add(dir={path=${config.path}},
ordinal=${config.ordinal})
```

   The **`mp-config.cli`** CLI script creates a new **`ConfigSource`**, to which ordinal and path values are retrieved from a properties file.

4. Save the script in the **`/scripts`** directory, which is located at the root directory of the project.
5. Add the following configuration extract to the existing plug-in **`<configuration>`** element:

```
<cli-sessions>
    <cli-session>
        <properties-file>
            scripts/cli.properties
        </properties-file>
        <script-files>
            <script>scripts/mp-config.cli</script>
        </script-files>
    </cli-session>
</cli-sessions>
```

6. Package the application:

```
$ mvn package
```

7. Log in to your OpenShift instance using the **`oc login`** command.

8. *Optional:* If you have not previously created a **`target/openshift`** subdirectory, you must create the suddirectory by issuing the following command:

```
$ mkdir target/openshift
```

9. Copy the packaged application into the created subdirectory.

```
$ cp target/microprofile-config-bootable.jar target/
openshift
```

10. Use the **`target/openshift`** subdirectory as the binary input to build the application:

```
$ oc start-build microprofile-config-app --from-dir
target/openshift
```

> **Note**
>
> OpenShift applies a set of CLI script commands to the bootable JAR configuration file to enable it for the cloud environment. You can access this script by opening the **`bootable-jar-build-artifacts/generated-cli-script.txt`** file in the Maven project **`/target`** directory.

11. Create a **`ConfigMap`**. For example:

```
$ oc create configmap microprofile-config-map --from-
literal=name="Name comes from Openshift ConfigMap"
```

12. Mount the **`ConfigMap`** into the application with the dc. For example:

```
$ oc set volume deployments/microprofile-config-app --add
--name=config-volume \
--mount-path=/etc/config \
--type=configmap \
--configmap-name=microprofile-config-map
```

After executing the **`oc set volume`** command, the application is re-deployed with the new configuration settings.

13. Test the output:

```
$ curl http://$(oc get route microprofile-config-app --
template='{{ .spec.host }}')/config/json
```

The following is the expected output:

```
{"result":"Hello Name comes from Openshift ConfigMap"}
```

Additional resources

- For information about MicroProfile Config **`ConfigSources`** attributes, see Default MicroProfile Config attributes.
- For information about bootable JAR arguments, see Supported bootable JAR arguments.

11.12. CREATING A BOOTABLE JAR MAVEN PROJECT

Follow the steps in the procedure to create an example Maven project. You must create a Maven project before you can perform the following procedures:

- Enabling JSON logging for your bootable JAR
- Enabling web session data storage for multiple bootable JAR instances
- Enabling HTTP authentication for bootable JAR with a CLI script
- Securing your JBoss EAP bootable JAR application with Red Hat build of Keycloak

In the project **`pom.xml`** file, you can configure Maven to retrieve the project artifacts required to build your bootable JAR.

Procedure

1. Set up the Maven project:

```
$ mvn archetype:generate \
-DgroupId=GROUP_ID \
-DartifactId=ARTIFACT_ID \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-webapp \
-DinteractiveMode=false
```

Where *GROUP_ID* is the **`groupId`** of your project and *ARTIFACT_ID* is the **`artifactId`** of your project.

2. In the **`pom.xml`** file, configure Maven to retrieve the JBoss EAP BOM file from a remote repository.

```
<repositories>
    <repository>
        <id>jboss</id>
        <url>https://maven.repository.redhat.com/ga</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
      <id>jboss</id>
        <url>https://maven.repository.redhat.com/ga</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
  </pluginRepository>
</pluginRepositories>
```

3. To configure Maven to automatically manage versions for the Jakarta EE artifacts in the **`jboss-eap-ee`** BOM, add the BOM to the **`<dependencyManagement>`** section of the project **`pom.xml`** file. For example:

```
<dependencyManagement>
  <dependencies>
    <dependency>
        <groupId>org.jboss.bom</groupId>
        <artifactId>jboss-eap-ee</artifactId>
        <version>8.0.2.GA-redhat-00007</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
```

```
  </dependencies>
</dependencyManagement>
```

4. Add the servlet API artifact, which is managed by the BOM, to the **<dependency>** section of the project **pom.xml** file, as shown in the following example:

```
<dependency>
    <groupId>jakarta.servlet</groupId>
    <artifactId>jakarta.servlet-api</artifactId>
    <scope>provided</scope>
</dependency>
```

Additional resources

- JBoss EAP XP Bootable JAR Maven Plugin.
- Specifying Galleon layers for your bootable JAR server.
- Securing your JBoss EAP bootable JAR application with Red Hat build of Keycloak.

11.13. ENABLING JSON LOGGING FOR YOUR BOOTABLE JAR

You can enable JSON logging for your bootable JAR by configuring the server logging configuration with a CLI script. When you enable JSON logging, you can use the JSON formatter to view log messages in JSON format.

The example in this procedure shows you how to enable JSON logging for your bootable JAR on a bare-metal platform and an OpenShift platform.

Prerequisites

- You have checked the latest Maven plug-in version, such as **9.minor.micro.Final-redhat-XXXXX**, where *9* is the major version, *minor* is the minor version, *micro* micro version and *X* is the Red Hat build number. For example: **9.0.1.Final-redhat-00009**.
- You have created a Maven project, and added dependencies for creating an application. See Creating a bootable JAR Maven project.

Important

In the Maven archetype of your Maven project, you must specify the groupID and artifactID that are specific to your project. For example:

```
$ mvn archetype:generate \
-DgroupId=com.example.logging \
-DartifactId=logging \
```

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

Note

The examples shown in the procedure specify the following properties:

- **`${bootable.jar.maven.plugin.version}`** for the Maven plug-in version.

You must set these properties in your project. For example:

```
<properties>
    <bootable.jar.maven.plugin.version>9.0.1.Final-redhat-00009</bootable.jar.maven.plugin.version>
</properties>
```

Procedure

1. Add the JBoss Logging and Jakarta RESTful Web Services dependencies, which are managed by the BOM, to the **`<dependencies>`** section of the project **`pom.xml`** file. For example:

```
<dependencies>
    <dependency>
        <groupId>org.jboss.logging</groupId>
        <artifactId>jboss-logging</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>jakarta.ws.rs</groupId>
        <artifactId>jakarta.ws.rs-api</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

2. Add the following content to the **`<build>`** element of the **`pom.xml`** file. For example:

```
<plugins>
        <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-jar-maven-plugin</artifactId>
```

```
<version>${bootable.jar.maven.plugin.version}</version>
                <configuration>
                        <channels>
                                <channel>
                                        <manifest>

<groupId>org.jboss.eap.channels</groupId>

<artifactId>eap-8.0</artifactId>
                                        </manifest>
                                </channel>
                                <channel>
                                        <manifest>

<groupId>org.jboss.eap.channels</groupId>

<artifactId>eap-xp-5.0</artifactId>
                                        </manifest>
                                </channel>
                        </channels>
                        <feature-packs>
                                <feature-pack>

<location>org.jboss.eap.xp:wildfly-galleon-pack</location>
                                </feature-pack>
                        </feature-packs>
                        <layers>
                                <layer>jaxrs-server</layer>
                        </layers>
                </configuration>
                <executions>
                        <execution>
                                <goals>
                                        <goal>package</
goal>
                                </goals>
                        </execution>
                </executions>
        </plugin>
</plugins>
```

3. Create the directory to store Java files:

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

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

4. Create a Java file **RestApplication.java** with the following content and save the file in the **APPLICATION_ROOT/src/main/java/com/example/logging/** directory:

```
package com.example.logging;
import jakarta.ws.rs.ApplicationPath;
import jakarta.ws.rs.core.Application;

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

5. Create a Java file **HelloWorldEndpoint.java** with the following content and save the file in the **APPLICATION_ROOT/src/main/java/com/example/logging/** directory:

```
package com.example.logging;

import jakarta.ws.rs.Path;
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Produces;

import org.jboss.logging.Logger;
@Path("/hello")
public class HelloWorldEndpoint {

    private static Logger log =
Logger.getLogger(HelloWorldEndpoint.class.getName());
    @GET
    @Produces("text/plain")
    public Response doGet() {
        log.debug("HelloWorldEndpoint.doGet called");
        return Response.ok("Hello from XP bootable
jar!").build();
    }
}
```

6. Create a CLI script, such as **logging.cli**, and save it in an accessible directory in the bootable JAR, such as the ***APPLICATION_ROOT*/scripts** directory, where **APPLICATION_ROOT** is the root directory of your Maven project. The script must contain the following commands:

```
/subsystem=logging/
logger=com.example.logging:add(level=ALL)
/subsystem=logging/json-formatter=json-
formatter:add(exception-output-type=formatted, pretty-
print=false, meta-data={version="1"}, key-
overrides={timestamp="@timestamp"})
/subsystem=logging/console-handler=CONSOLE:write-
attribute(name=level,value=ALL)
/subsystem=logging/console-handler=CONSOLE:write-
attribute(name=named-formatter, value=json-formatter)
```

7. Add the following configuration extract to the plug-in **`<configuration>`** element:

```
<cli-sessions>
        <cli-session>
        <script-files>
            <script>scripts/logging.cli</script>
        </script-files>
    </cli-session>
</cli-sessions>
```

This example shows the **`logging.cli`** CLI script, which modifies the server logging configuration file to enable JSON logging for your application.

8. Package the application as a bootable JAR.

```
$ mvn package
```

9. *Optional*: To run the application on a JBoss EAP bare-metal platform, follow the steps outlined in Using a bootable JAR on a JBoss EAP bare-metal platform, but with the following difference:

    a. Start the application:

    ```
    mvn wildfly-jar:run
    ```

    b. Verification: You can access the application by specifying the following URL in your browser: http://127.0.0.1:8080/hello.

    Expected output: You can view the JSON-formatted logs, including the **`com.example.logging.HelloWorldEndpoint`** debug trace, in the application console.

10. *Optional*: To run the application on a JBoss EAP OpenShift platform, complete the following steps:

    a. Add the **`<cloud/>`** element to the plug-in configuration. For example:

```
<plugins>
   <plugin>
       ... <!-- You must evolve the existing
configuration with the <cloud/> element  -->
      <configuration >
          ...
          <cloud/>
       </configuration>
    </plugin>
</plugins>
```

b. Rebuild the application:

```
$ mvn clean package
```

c. Log in to your OpenShift instance using the **`oc login`** command.

d. Create a new project in OpenShift. For example:

```
$ oc new-project bootable-jar-project
```

e. Enter the following **`oc`** commands to create an application image:

```
$ mkdir target/openshift && cp target/logging-
bootable.jar target/openshift  1

$ oc import-image ubi8/openjdk-17 --
from=registry.redhat.io/ubi8/openjdk-17 --confirm
  2

$ oc new-build --strategy source --binary --image-
stream openjdk-17 --name logging  3

$ oc start-build logging --from-dir target/openshift
 4
```

Creates the **`target/openshift`** subdirectory. The packaged application is copied into the **`openshift`** subdirectory.

Imports the latest OpenJDK 17 imagestream tag and image information into the OpenShift project.

Creates a build configuration based on the logging directory and the OpenJDK 17 imagestream.

Uses the **`target/openshift`** subdirectory as the binary input to build the application.

f. Deploy the application:

```
$ oc new-app logging

$ oc expose svc/logging
```

g. Get the URL of the route.

```
$ oc get route logging --template='{{ .spec.host }}'
```

h. Access the application in your web browser using the URL returned from the previous command. For example:

```
http://ROUTE_NAME/hello
```

i. *Verification:* Issue the following command to view a list of OpenShift pods available, and to check the pods build statuses:

```
$ oc get pods
```

Access a running pod log of your application. Where **`APP_POD_NAME`** is the name of the running pod logging application.

```
$ oc logs APP_POD_NAME
```

Expected outcome: The pod log is in JSON format and includes the **`com.example.logging.HelloWorldEndpoint`** debug trace.

Additional resources

- For information about using a bootable JAR on OpenShift, see Using a bootable JAR on a JBoss EAP OpenShift platform.
- For information about specifying the JBoss EAP JAR Maven for your project, see Specifying Galleon layers for your bootable JAR server.
- For information about creating CLI scripts, see CLI scripts.

11.14. ENABLING WEB SESSION DATA STORAGE FOR MULTIPLE BOOTABLE JAR INSTANCES

You can build and package a web-clustering application as a bootable JAR.

Prerequisites

- You have checked the latest Maven plug-in version, such as **`9.minor.micro.Final-redhat-XXXXX`**, where *9* is the major version, *minor* is the minor version, *micro* micro version and *X* is the Red Hat build number. For example: **`9.0.1.Final-redhat-00009`**.
- You have created a Maven project, and added dependencies for creating a web-clustering application. See Creating a bootable JAR Maven project.

> **Important**
>
> When setting up the Maven project, you must specify values in the Maven archetype configuration. For example:

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

> **Note**
>
> The examples shown in the procedure specify the following properties:
>
> - **`${bootable.jar.maven.plugin.version}`** for the Maven plug-in version.
>
> You must set these properties in your project. For example:

```
<properties>
    <bootable.jar.maven.plugin.version>9.0.1.Final-redhat-00009</bootable.jar.maven.plugin.version>
</properties>
```

Procedure

1. Add the following content to the **`<build>`** element of the **`pom.xml`** file. For example:

```
<plugins>
        <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-jar-maven-plugin</
artifactId>

<version>${bootable.jar.maven.plugin.version}</version>
                <configuration>
                        <channels>
                                <channel>
                                        <manifest>

<groupId>org.jboss.eap.channels</groupId>

<artifactId>eap-8.0</artifactId>
                                        </manifest>
                                </channel>
                                <channel>
                                        <manifest>

<groupId>org.jboss.eap.channels</groupId>

<artifactId>eap-xp-5.0</artifactId>
                                        </manifest>
                                </channel>
                        </channels>
                        <feature-pack-
location>org.jboss.eap.xp:wildfly-galleon-pack</feature-
pack-location>
                        <layers>
                                <layer>datasources-web-
server</layer>
                                <layer>web-clustering</
layer>
                        </layers>
                </configuration>
                <executions>
                        <execution>
                                <goals>
                                        <goal>package</
goal>
                                </goals>
                        </execution>
                </executions>
```

```
        </plugin>
</plugins>
```

Note

This example makes use of the **`web-clustering`** Galleon layer to enable web session sharing.

2. Update the **`web.xml`** file in the **`src/main/webapp/WEB-INF`** directory with the following configuration:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="4.0"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/
javaee  http://xmlns.jcp.org/xml/ns/javaee/web-
app_4_0.xsd">
    <distributable/>
</web-app>
```

The **`<distributable/>`** tag indicates that this servlet can be distributed across multiple servers.

3. Create the directory to store Java files:

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

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

4. Create a Java file **`MyServlet.java`** with the following content and save the file in the ***`APPLICATION_ROOT`*`/src/main/java/com/example/webclustering/`** directory.

```
package com.example.webclustering;

import java.io.IOException;
import java.io.PrintWriter;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
```

```
@WebServlet(urlPatterns = {"/clustering"})
public class MyServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws IOException {
        response.setContentType("text/html;charset=UTF-8");
        long t;
        User user = (User)
request.getSession().getAttribute("user");
        if (user == null) {
            t = System.currentTimeMillis();
            user = new User(t);
            request.getSession().setAttribute("user",
user);
        }
        try (PrintWriter out = response.getWriter()) {
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Web clustering demo</
title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Session id " +
request.getSession().getId() + "</h1>");
            out.println("<h1>User Created " +
user.getCreated() + "</h1>");
            out.println("<h1>Host Name " +
System.getenv("HOSTNAME") + "</h1>");
            out.println("</body>");
            out.println("</html>");
        }
    }
}
```

The content in **`MyServlet.java`** defines the endpoint to which a client sends an HTTP request.

5. Create a Java file **`User.java`** with the following content and save the file in the ***`APPLICATION_ROOT`*`/src/main/java/com/example/webclustering/`** directory.

```
package com.example.webclustering;

import java.io.Serializable;
```

```
public class User implements Serializable {
    private final long created;

    User(long created) {
        this.created = created;
    }
    public long getCreated() {
        return created;
    }
}
```

6. Package the application:

```
$ mvn package
```

7. *Optional*: To run the application on a JBoss EAP bare-metal platform, follow the steps outlined in Using a bootable JAR on a JBoss EAP bare-metal platform, but with the following difference:

    a. On a JBoss EAP bare-metal platform, you can use the **`java -jar`** command to run multiple bootable JAR instances, as demonstrated in the following examples:

```
$ java -jar target/web-clustering-bootable.jar -
Djboss.node.name=node1

$ java -jar target/web-clustering-bootable.jar -
Djboss.node.name=node2 -Djboss.socket.binding.port-
offset=10
```

    b. *Verification*: You can access the application on the node 1 instance: http://127.0.0.1:8080/clustering. Note the user session ID and the user-creation time.

    After you kill this instance, you can access the node 2 instance: http://127.0.0.1:8090/clustering. The user must match the session ID and the user-creation time of the node 1 instance.

8. *Optional*: To run the application on a JBoss EAP OpenShift platform, follow the steps outlined in Using a bootable JAR on a JBoss EAP OpenShift platform, but complete the following steps:

    a. Add the **`<cloud/>`** element to the plug-in configuration. For example:

```
<plugins>
   <plugin>
       ... <!-- You must evolve the existing
configuration with the <cloud/> element  -->
       <configuration >
```

```
            ...
            <cloud/>
        </configuration>
    </plugin>
</plugins>
```

b. Rebuild the application:

```
$ mvn clean package
```

c. Log in to your OpenShift instance using the **`oc login`** command.

d. Create a new project in OpenShift. For example:

```
$ oc new-project bootable-jar-project
```

e. To run a web-clustering application on a JBoss EAP OpenShift platform, authorization access must be granted for the service account that the pod is running in. The service account can then access the Kubernetes REST API. The following example shows authorization access being granted to a service account:

```
$ oc policy add-role-to-user view
system:serviceaccount:$(oc project -q):default
```

f. Enter the following **`oc`** commands to create an application image:

```
$ mkdir target/openshift && cp target/web-clustering-
bootable.jar target/openshift  1

$ oc import-image ubi8/openjdk-17 --
from=registry.redhat.io/ubi8/openjdk-17 --confirm  2

$ oc new-build --strategy source --binary --image-
stream openjdk-17 --name web-clustering  3

$ oc start-build web-clustering --from-dir target/
openshift  4
```

Creates the **`target/openshift`** sub-directory. The packaged application is copied into the **`openshift`** sub-directory.

Imports the latest OpenJDK 17 imagestream tag and image information into the OpenShift project.

Creates a build configuration based on the web-clustering directory and the OpenJDK 17 imagestream.

Uses the **`target/openshift`** sub-directory as the binary input to build the application.

g. Deploy the application:

```
$ oc new-app web-clustering -e
KUBERNETES_NAMESPACE=$(oc project -q)

$ oc expose svc/web-clustering
```

Important

You must use the **`KUBERNETES_NAMESPACE`** environment variable to view other pods in the current OpenShift namespace; otherwise, the server attempts to retrieve the pods from the **`default`** namespace.

h. Get the URL of the route.

```
$ oc get route web-clustering --template='{{
.spec.host }}'
```

i. Access the application in your web browser using the URL returned from the previous command. For example:

```
http://ROUTE_NAME/clustering
```

Note the user session ID and user creation time.

j. Scale the application to two pods:

```
$ oc scale --replicas=2 deployments web-clustering
```

k. Issue the following command to view a list of OpenShift pods available, and to check the pods build statuses:

```
$ oc get pods
```

l. Kill the oldest pod using the **`oc delete pod web-clustering-POD_NAME`** command, where *POD_NAME* is the name of your oldest pod.

m. Access the application again:

```
http://ROUTE_NAME/clustering
```

Expected outcome: The session ID and the creation time generated by the new pod match those of the of the terminated pod. This indicates that web session data storage is enabled.

Additional resources

- For information about distributable web session management profiles, see The distributable-web subsystem for Distributable Web Session Configurations in the *Development Guide*.
- For information about configuring the JGroups protocol stack, see Configuring a JGroups Discovery Mechanism in the *Getting Started with JBoss EAP for OpenShift Container Platform* guide.

11.15. ENABLING HTTP AUTHENTICATION FOR BOOTABLE JAR WITH A CLI SCRIPT

You can enable HTTP authentication for the bootable JAR with a CLI script. This script adds a security realm and a security domain to your server.

Prerequisites

- You have checked the latest Maven plug-in version, such as **`9.minor.micro.Final-redhat-XXXXX`**, where *9* is the major version, *minor* is the minor version, *micro* micro version and *X* is the Red Hat build number. For example: **`9.0.1.Final-redhat-00009`**.
- You have created a Maven project, and added dependencies for creating an application that requires HTTP authentication. See Creating a bootable JAR Maven project.

Important

When setting up the Maven project, you must specify HTTP authentication values in the Maven archetype configuration. For example:

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

Note

The examples shown in the procedure specify the following properties:

- **`${bootable.jar.maven.plugin.version}`** for the Maven plug-in version.

You must set these properties in your project. For example:

```
<properties>
    <bootable.jar.maven.plugin.version>9.0.1.Final-redhat-00009</bootable.jar.maven.plugin.version>
</properties>
```

Procedure

1. Add the following content to the **`<build>`** element of the **`pom.xml`** file. For example:

```
<plugins>
        <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-jar-maven-plugin</artifactId>

<version>${bootable.jar.maven.plugin.version}</version>
                <configuration>
                        <channels>
                                <channel>
                                        <manifest>

<groupId>org.jboss.eap.channels</groupId>

<artifactId>eap-8.0</artifactId>
                                        </manifest>
                                </channel>
                                <channel>
```

```
                                                <manifest>

<groupId>org.jboss.eap.channels</groupId>

<artifactId>eap-xp-5.0</artifactId>
                                                </manifest>
                                        </channel>
                                </channels>
                                <feature-pack-
location>org.jboss.eap.xp:wildfly-galleon-pack</feature-
pack-location>
                                <layers>
                                        <layer>datasources-web-
server</layer>
                                </layers>
                        </configuration>
                        <executions>
                                <execution>
                                        <goals>
                                                <goal>package</
goal>
                                        </goals>
                                </execution>
                        </executions>
                </plugin>
</plugins>
```

The example shows the inclusion of the **`datasources-web-server`** Galleon layer that contains the **`elytron`** subsystem.

2. Update the **`web.xml`** file in the **`src/main/webapp/WEB-INF`** directory. For example:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="4.0"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/
javaee  http://xmlns.jcp.org/xml/ns/javaee/web-
app_4_0.xsd">

    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>Example Realm</realm-name>
```

```
    </login-config>

</web-app>
```

3. Create the directory to store Java files:

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

Where **`APPLICATION_ROOT`** is the root directory of your Maven project.

4. Create a Java file **`TestServlet.java`** with the following content and save the file in the **`APPLICATION_ROOT/src/main/java/com/example/authentication/`** directory.

```
package com.example.authentication;

import jakarta.servlet.annotation.HttpMethodConstraint;
import jakarta.servlet.annotation.ServletSecurity;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.io.PrintWriter;

@WebServlet(urlPatterns = "/hello")
@ServletSecurity(httpMethodConstraints = {
@HttpMethodConstraint(value = "GET", rolesAllowed = {
"Users" }) })
public class TestServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req,
HttpServletResponse resp) throws IOException {
        PrintWriter writer = resp.getWriter();
        writer.println("Hello " +
req.getUserPrincipal().getName());
        writer.close();
    }

}
```

5. Create a CLI script, such as **`authentication.cli`**, and save it in an accessible directory in the bootable JAR,

such as the ***APPLICATION_ROOT*/scripts** directory. The script must contain the following commands:

```
/subsystem=elytron/properties-realm=bootable-
realm:add(users-properties={relative-
to=jboss.server.config.dir, path=bootable-
users.properties, plain-text=true}, groups-
properties={relative-to=jboss.server.config.dir,
path=bootable-groups.properties})
/subsystem=elytron/security-
domain=BootableDomain:add(default-realm=bootable-realm,
permission-mapper=default-permission-mapper,
realms=[{realm=bootable-realm, role-decoder=groups-to-
roles}])

/subsystem=undertow/application-security-
domain=other:write-attribute(name=security-domain,
value=BootableDomain)
```

6. Add the following configuration extract to the plug-in **<configuration>** element:

```
<cli-sessions>
    <cli-session>
        <script-files>
            <script>scripts/authentication.cli</script>
        </script-files>
    </cli-session>
</cli-sessions>
```

This example shows the **authentication.cli** CLI script, which configures the default **undertow** security domain to the security domain defined for your server.

> **Note**
>
> You have the option to execute the CLI script at runtime instead of packaging time. To do so, skip this step and proceed to step 10.

7. In the root directory of your Maven project create a directory to store the properties files that the JBoss EAP JAR Maven plug-in adds to the bootable JAR:

```
$ mkdir -p APPLICATION_ROOT/extra-content/standalone/
configuration/
```

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

This directory stores files such as **`bootable-users.properties`** and **`bootable-groups.properties`** files.

The **`bootable-users.properties`** file contains the following content:

```
testuser=bootable_password
```

The **`bootable-groups.properties`** file contains the following content:

```
testuser=Users
```

8. Add the following **`extra-content-content-dirs`** element to the existing **`<configuration>`** element:

```
<extra-server-content-dirs>
            <extra-content>extra-content</extra-content>
</extra-server-content-dirs>
```

The **`extra-content`** directory contains the properties files.

9. Package the application as a bootable JAR.

```
$ mvn package
```

10. Start the application:

```
mvn wildfly-jar:run
```

If you have chosen to skip step 6 and not execute the CLI script during build, launch the application with the following command:

```
mvn wildfly-jar:run -Dwildfly.bootable.arguments=--cli-
script=scripts/authentication.cli
```

11. Call the servlet, but do not specify credentials:

```
curl -v http://localhost:8080/hello
```

Expected output:

```
HTTP/1.1 401 Unauthorized
...
WWW-Authenticate: Basic realm="Example Realm"
```

12. Call the server and specify your credentials. For example:

```
$ curl -v -u testuser:bootable_password http://
localhost:8080/hello
```

A HTTP 200 status is returned that indicates HTTP authentication is enabled for your bootable JAR. For example:

```
HTTP/1.1 200 OK
....
Hello testuser
```

Additional resources

- For information about enabling HTTP authentication for the **`undertow`** security domain, see Enable HTTP Authentication for Applications Using the CLI Security Command in the *How to Configure Server Security*.

CHAPTER 12. OBSERVABILITY IN JBOSS EAP

If you're a developer or system administrator, *observability* is a set of practices and technologies you can use to determine, based on certain signals from your application, the location and source of a problem in your application. The most common signals are metrics, events, and tracing. JBoss EAP uses OpenTelemetry for *observability*.

12.1. OPENTELEMETRY IN JBOSS EAP

OpenTelemetry is a set of tools, application programming interfaces (APIs), and software development kits (SDKs) you can use to instrument, generate, collect, and export telemetry data for your applications. Telemetry data includes metrics, logs, and traces. Analyzing an application's telemetry data helps you to improve your application's performance. JBoss EAP provides OpenTelemetry capability through the **`opentelemetry`** subsystem.

Note

Red Hat JBoss Enterprise Application Platform 8.0 provides only OpenTelemetry tracing capabilities.

Important

OpenTelemetry is a Technology Preview feature only. Technology Preview features are not supported with Red

Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview .

Additional resources

- OpenTelemetry Documentation

12.2. OPENTELEMETRY CONFIGURATION IN JBOSS EAP

You configure a number of aspects of OpenTelemetry in JBoss EAP using the **`opentelemetry`** subsystem. These include exporter, span processor, and sampler.

exporter

To analyze and visualize traces and metrics, you export them to a collector such as Jaeger. You can configure JBoss EAP to use either Jaeger or any collector that supports the OpenTelemetry protocol (OTLP).

span processor

You can configure the span processor to export spans either as they are produced or in batches. You can also configure the number of traces to export.

sampler

You can configure the number of traces to record by configuring the sampler.

Example configuration

The following XML is an example of the full OpenTelemetry configuration, including default values. JBoss EAP does not persist the default values when you make changes, so your configuration might look different.

```
<subsystem xmlns="urn:wildfly:opentelemetry:1.0"
        service-name="example">
    <exporter
        type="jaeger"
        endpoint="http://localhost:14250"/>
    <span-processor
        type="batch"
        batch-delay="4500"
        max-queue-size="128"
        max-export-batch-size="512"
        export-timeout="45"/>
    <sampler
```

```
        type="on"/>
</subsystem>
```

Note

You cannot use an OpenShift route object to connect with a Jaeger endpoint. Instead, use **`http://<ip_address>:<port>`** or **`http://<service_name>:<port>`**.

Additional resources

- OpenTelemetry subsystem attributes

12.3. OPENTELEMETRY TRACING IN JBOSS EAP

JBoss EAP provides OpenTelemetry tracing to help you track the progress of user requests as they pass through different parts of your application. By analyzing traces, you can improve your application's performance and debug availability issues.

OpenTelemetry tracing consists of the following components:

Trace

A collection of operations that a request goes through in an application.

Span

A single operation within a trace. It provides request, error, and duration (RED) metrics and contains a span context.

Span context

A set of unique identifiers that represents a request that the containing span is a part of.

JBoss EAP automatically traces REST calls to your Jakarta RESTful Web Services applications and container-managed Jakarta RESTful Web Services client invocations. JBoss EAP traces REST calls implicitly as follows:

- For each incoming request:
  - JBoss EAP extracts the span context from the request.
  - JBoss EAP starts a new span, then closes it when the request is completed.
- For each outgoing request:
  - JBoss EAP injects span context into the request.
  - JBoss EAP starts a new span, then closes it when the request is completed.

In addition to implicit tracing, you can create custom spans by injecting a **`Tracer`** instance into your application for granular tracing.

Additional resources

- Using Jaeger to observe the OpenTelemetry traces for an application
- OpenTelemetry application development in JBoss EAP

12.4. ENABLING OPENTELEMETRY TRACING IN JBOSS EAP

To use OpenTelemetry tracing in JBoss EAP you must first enable the **`opentelemetry`** subsystem.

Prerequisites

- JBoss EAP 8.0 with JBoss EAP XP 5.0 is installed.

Procedure

1. Add the OpenTelemetry extension using the management CLI.

```
/extension=org.wildfly.extension.opentelemetry:add
```

2. Enable the **`opentelemetry`** subsystem using the management CLI.

```
/subsystem=opentelemetry:add
```

3. Reload JBoss EAP.

```
reload
```

Additional resources

- Configuring the **`opentelemetry`** subsystem

12.5. CONFIGURING THE OPENTELEMETRY SUBSYSTEM

You can configure the **`opentelemetry`** subsystem to set different aspects of tracing. Configure these based on the collector you use for observing the traces.

Prerequisites

- You have enabled the **`opentelemetry`** subsystem. For more information, see Enabling OpenTelemetry tracing in JBoss EAP.

Procedure

1. Set the exporter type for the traces.

Syntax

```
/subsystem=opentelemetry:write-attribute(name=exporter-type, value=<exporter_type>)
```

Example

```
/subsystem=opentelemetry:write-attribute(name=exporter-type, value=jaeger)
```

2. Set the endpoint at which to export the traces.

Syntax

```
/subsystem=opentelemetry:write-attribute(name=endpoint, value=<URL:port>)
```

Example

```
/subsystem=opentelemetry:write-attribute(name=endpoint, value=http:localhost:14250)
```

3. Set the service name under which the traces are exported.

Syntax

```
/subsystem=opentelemetry:write-attribute(name=service-name, value=<service_name>)
```

Example

```
/subsystem=opentelemetry:write-attribute(name=service-name, value=exampleOpenTelemetryService)
```

Additional resources

- Using Jaeger to observe the OpenTelemetry traces for an application

12.6. USING JAEGER TO OBSERVE THE OPENTELEMETRY

TRACES FOR AN APPLICATION

JBoss EAP automatically and implicitly traces REST calls to Jakarta RESTful Web Services applications. You do not need to add any configuration to your Jakarta RESTful Web Services application or configure the **`opentelemetry`** subsystem. The following procedure demonstrates how to observe traces for the **`helloworld-rs`** quickstart in the Jaeger console.

Prerequisites

- You have installed Docker. For more information, see Get Docker.
- You have downloaded the **`helloworld-rs`** quickstart. The quickstart is available at helloworld-rs.
- You have configured the the **`opentelemetry`** subsystem. For more information, see Configuring the **`opentelemetry`** subsystem.

Procedure

1. Start the Jaeger console using its Docker image.

```
$ docker run -d --name jaeger \
  -e COLLECTOR_ZIPKIN_HOST_PORT=:9411 \
  -p 5775:5775/udp \
  -p 6831:6831/udp \
  -p 6832:6832/udp \
  -p 5778:5778 \
  -p 16686:16686 \
  -p 14268:14268 \
  -p 14250:14250 \
  -p 9411:9411 \
  jaegertracing/all-in-one:1.29
```

2. Use Maven to deploy the **`helloworld-rs`** quickstart from its root directory.

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

3. In a web browser, access the quickstart at **`http://localhost:8080/helloworld-rs/`**, then click any link.
4. In a web browser, open the Jaeger console at **`http://localhost:16686/search`**. **`hello-world.rs`** is listed under **Service**.
5. Select **`hello-world.rs`** and click **Find Traces**. The details of the trace for **`hello-world.rs`** are listed.

Additional resources

- OpenTelemetry application development in JBoss EAP

12.7. OPENTELEMETRY TRACING APPLICATION DEVELOPMENT

Although JBoss EAP automatically and implicitly traces REST calls to Jakarta RESTful Web Services applications, you can create custom spans from your application for granular tracing. A *span* is a single operation within a trace. You can create a span when, for example, a resource is defined, a method is called, and so on, in your application. You create custom traces in your application by injecting a **`Tracer`** instance.

12.7.1. Configuring a Maven project for OpenTelemetry tracing

For creating an OpenTelemetry tracing application, create a Maven project with the required dependencies and directory structure.

Prerequisites

- You have installed Maven. For more information, see Downloading Apache Maven.
- You have configured your Maven repository for the latest release. For information about installing the latest Maven repository patch, see Maven and the JBoss EAP microprofile maven repository.

Procedure

1. In the CLI, use the **`mvn`** command to set up a Maven project. This command creates the directory structure for the project and the **`pom.xml`** configuration file.

   Syntax

   ```
   $ mvn archetype:generate \
   -DgroupId=<group-to-which-your-application-belongs> \
   -DartifactId=<name-of-your-application> \
   -DarchetypeGroupId=org.apache.maven.archetypes \
   -DarchetypeArtifactId=maven-archetype-webapp \
   -DinteractiveMode=false
   ```

   Example

   ```
   $ mvn archetype:generate \
   -DgroupId=com.example.opentelemetry \
   -DartifactId=simple-tracing-example \
   -DarchetypeGroupId=org.apache.maven.archetypes \
   -DarchetypeArtifactId=maven-archetype-webapp \
   -DinteractiveMode=false
   ```

2. Navigate to the application root directory.

Syntax

```
$ cd <name-of-your-application>
```

Example

```
$ cd simple-tracing-example
```

3. Update the generated **`pom.xml`** file.

a. Set the following properties:

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <failOnMissingWebXml>false</failOnMissingWebXml>
    <version.server.bom>4.0.0.GA</version.server.bom>
    <version.wildfly-jar.maven.plugin>6.1.1.Final</
version.wildfly-jar.maven.plugin>
</properties>
```

b. Set the following dependencies:

```
<dependencies>
    <dependency>
        <groupId>jakarta.enterprise</groupId>
        <artifactId>jakarta.enterprise.cdi-api</
artifactId>
        <version>2.0.2</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.jboss.spec.javax.ws.rs</groupId>
        <artifactId>jboss-jaxrs-api_2.1_spec</
artifactId>
        <version>2.0.2.Final</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>io.opentelemetry</groupId>
        <artifactId>opentelemetry-api</artifactId>
```

```
            <version>1.5.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
```

c. Set the following build configuration to use **`mvn widlfy:deploy`** to deploy the application:

```
<build>
  <!-- Set the name of the archive -->
  <finalName>${project.artifactId}</finalName>
  <plugins>
    <!-- Allows to use mvn wildfly:deploy -->
    <plugin>
      <groupId>org.wildfly.plugins</groupId>
      <artifactId>wildfly-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Verification

- In the application root directory, enter the following command:

```
$ mvn install
```

You get an output similar to the following:

```
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 1.440 s
[INFO] Finished at: 2021-12-27T14:45:12+05:30
[INFO]
------------------------------------------------------------------------
```

You can now create an OpenTelemetry tracing application.

Additional resources

- Creating applications that create custom spans

12.7.2. Creating applications that create custom spans

The following procedure demonstrates how to create an application that can create two custom spans like these:

- **`prepare-hello`** - When the method **`getHello()`** in the application is called.
- **`process-hello`** - When the value **`hello`** is assigned to a new **`String`** object **`hello`**.

This procedure also demonstrates how to view these spans in a Jaeger console. **`<application_root>`** in the procedure denotes the directory that contains the **`pom.xml`** file, which contains the Maven configuration for your application.

Prerequisites

- You have installed Docker. For more information, see Get Docker.
- You have created a Maven project. For more information, see Configuring Maven project for OpenTelemetry tracing.
- You have configured the the **`opentelemetry`** subsystem. For more information, see Configuring the **`opentelemetry`** subsystem.

Procedure

1. In the *<application_root>*, create a directory to store the Java files.

   Syntax

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

   Example

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

2. Navigate to the new directory.

   Syntax

   ```
   $ cd src/main/java/com/example/opentelemetry
   ```

   Example

   ```
   $ cd src/main/java/com/example/opentelemetry
   ```

3. Create a **`JakartaRestApplication.java`** file with the following content. This **`JakartaRestApplication`** class declares the application as a Jakarta RESTful Web Services application.

   ```
   package com.example.opentelemetry;
   ```

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

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

4. Create an **`ExplicitlyTracedBean.java`** file with the following content for the class **`ExplicitlyTracedBean`**. This class creates custom spans by injecting a **`Tracer`** class.

```
package com.example.opentelemetry;

import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.Tracer;

@RequestScoped
public class ExplicitlyTracedBean {

    @Inject
    private Tracer tracer;  1

    public String getHello() {
        Span prepareHelloSpan =
tracer.spanBuilder("prepare-hello").startSpan();  2
        prepareHelloSpan.makeCurrent();

        String hello = "hello";

        Span processHelloSpan =
tracer.spanBuilder("process-hello").startSpan();  3
        processHelloSpan.makeCurrent();

        hello = hello.toUpperCase();

        processHelloSpan.end();
        prepareHelloSpan.end();

        return hello;
    }
}
```

Inject a **`Tracer`** class to create custom spans.

Create a span called **`prepare-hello`** to indicate that the method **`getHello()`** was called.

Create a span called **`process-hello`** to indicate that the value **`hello`** was assigned to a new **`String`** object called **`hello`**.

5. Create a **`TracedResource.java`** file with the following content for **`TracedResource`** class. This file injects the **`ExplicitlyTracedBean`** class and declares two endpoints: **`traced`** and **`cdi-trace`**.

```
package com.example.opentelemetry;

import javax.enterprise.context.RequestScoped;
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;

@Path("/hello")
@RequestScoped
public class TracedResource {
    @Inject
    private ExplicitlyTracedBean tracedBean;

    @GET
    @Path("/traced")
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }

    @GET
    @Path("/cdi-trace")
    @Produces(MediaType.TEXT_PLAIN)
    public String cdiHello() {
        return tracedBean.getHello();
    }
}
```

6. Navigate to the application root directory.

Syntax

```
$ cd <path_to_application_root>/<application_root>
```

Example

```
$ cd ~/applications/simple-tracing-example
```

7. Compile and deploy the application with the following command:

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

8. Start the Jaeger console.

```
$ docker run -d --name jaeger \
  -e COLLECTOR_ZIPKIN_HOST_PORT=:9411 \
  -p 5775:5775/udp \
  -p 6831:6831/udp \
  -p 6832:6832/udp \
  -p 5778:5778 \
  -p 16686:16686 \
  -p 14268:14268 \
  -p 14250:14250 \
  -p 9411:9411 \
  jaegertracing/all-in-one:1.29
```

9. In a browser, navigate to **`localhost:8080/simple-tracing-example/hello/cdi-trace`**.

10. In a browser, open the Jaeger console at **`http://localhost:16686/search`**.

11. In the Jaeger console, select **JBoss EAP XP** and click **Find Traces**.

12. Click **3 Spans**.

13. The Jaeger console displays the following traces:

```
|GET /hello/cdi-trace  1
-
 | prepare-hello  2
  -
   | process-hello  3
```

This is the span for the automatic implicit trace.

The custom span **`prepare-hello`** indicates that the method **`getHello()`** was called. It is the child of span for the automatic implicit trace.

The custom span **`process-hello`** indicates that the value **`hello`** was assigned to a new **`String`** object **`hello`**. It is the child of the **`prepare-hello`** span.

Whenever you access the application endpoint at **`http://localhost:16686/search`**, a new trace is created with all the child spans.

Additional resources

- OpenTelemetry tracing in JBoss EAP

CHAPTER 13. REFERENCE

13.1. MICROPROFILE CONFIG REFERENCE

13.1.1. Default MicroProfile Config attributes

The 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 13.1. Default MicroProfile Config attributes

`ConfigSource`	Ordinal
System properties	**`400`**
Environment variables	**`300`**

ConfigSource	Ordinal
Property files **META-INF/microprofile-config.properties** found on the classpath	**100**

13.1.2. MicroProfile Config SmallRye ConfigSources

The **microprofile-config-smallrye** project defines more **ConfigSources** you can use in addition to the default MicroProfile Config **ConfigSources**.

Table 13.2. Additional 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 MicroProfile Config specification.

13.2. MICROPROFILE FAULT TOLERANCE REFERENCE

13.2.1. MicroProfile Fault Tolerance configuration properties

SmallRye Fault Tolerance specification defines the following properties in addition to the properties defined in the MicroProfile Fault Tolerance specification.

Table 13.3. MicroProfile Fault Tolerance configuration properties

Property	Default value	Description

Property	Default value	Description
io.smallrye.faulttolerance.mainThreadPoolSize	**100**	Maximum number of threads in the thread pool.
io.smallrye.faulttolerance.mainThreadPoolQueueSize	**-1 (unbounded)**	Size of the queue that the thread pool should use.

13.3. MICROPROFILE JWT REFERENCE

13.3.1. MicroProfile Config JWT standard properties

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

Table 13.4. 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
```

13.4. MICROPROFILE OPENAPI REFERENCE

13.4.1. MicroProfile OpenAPI configuration properties

In addition to the standard MicroProfile OpenAPI configuration properties, JBoss EAP supports the following additional MicroProfile OpenAPI properties. These properties can be applied in both the global and the application scope.

Table 13.5. MicroProfile OpenAPI properties in JBoss EAP

Property	Default value	Description
`mp.openapi.extensions.enabled`	`true`	Enables or disables registration of an OpenAPI endpoint. When set to `false`, disables generation of OpenAPI documentation. You can set the value globally using the config subsystem, or for each application in a configuration file such as `/META-INF/microprofile-config.properties`. You can parameterize this property to selectively enable or disable `microprofile-openapi-smallrye` in different environments, such as production or development. You can use this property to control which application associated with a given virtual host should generate a MicroProfile OpenAPI model.
`mp.openapi.extensions.path`	`/openapi`	You can use this property for generating OpenAPI documentation for multiple applications associated with a virtual host. Set a distinct `mp.openapi.extensions.path` on each application associated with the same virtual host.

Property	Default value	Description
mp.openapi.extensions.servers.relative	**true**	Indicates whether auto-generated server records are absolute or relative to the location of the OpenAPI endpoint. Server records are necessary to ensure, in the presence of a non-root context path, that consumers of an OpenAPI document can construct valid URLs to REST services relative to the host of the OpenAPI endpoint. The value **true** indicates that the server records are relative to the location of the OpenAPI endpoint. The generated record contains the context path of the deployment. When set to **false**, JBoss EAP XP generates server records including all the protocols, hosts, and ports at which the deployment is accessible.

13.5. MICROPROFILE REACTIVE MESSAGING REFERENCE

13.5.1. MicroProfile reactive messaging connectors for integrating with external messaging systems

The following is a list of reactive messaging property key prefixes required by the MicroProfile Config specification:

- **mp.messaging.incoming.[channel-name].[attribute]=[value]**
- **mp.messaging.outgoing.[channel-name].[attribute]=[value]**
- **mp.messaging.connector.[connector-name].[attribute]=[value]**

Note that **channel-name** is either the **@Incoming.value()** or the **@Outgoing.value()**. For clarification, look at this example of a pair of connector methods:

```
@Outgoing("to")
public int send() {
   int i = // Randomly generated...
```

```
   return i;
}

@Incoming("from")
public void receive(int i) {
   // Process payload
}
```

In this example, the required property prefixes are as follows:

- **`mp.messaging.incoming.from`**. This defines the **`receive()`** method.
- **`mp.messaging.outgoing.to`**. This defines the **`send()`** method.

Remember that this is an example. Because different connectors recognize different properties, the prefixes you indicate depend on the connector you want to configure.

13.5.2. Example of the data exchange between reactive messaging streams and user-initialized code

The following is an example of data exchange between reactive messaging streams and code that a user triggered through the **`@Channel`** and **`Emitter`** constructs:

```
@Path("/")
@ApplicationScoped
class MyBean {
    @Inject @Channel("my-stream")
    Emitter<String> emitter;  1

    Publisher<String> dest;

    public MyBean() {  2
    }

    @Inject
    public MyBean(@Channel("my-stream") Publisher<String>
dest) {
        this.dest =
subscribeAndAllowMultipleSubscriptions(dest);
    }

    private Publisher
subscribeAndAllowMultipleSubscriptions(Publisher delegate) {
    }  3   4   5
```

```
    @POST
    public PublisherBuilder<String>
publish(@FormParam("value") String value) {
        return emitter.send(value);
    }

    @GET
    public Publisher poll() {
        return dest;
    }

    @PreDestroy
    public void close() {  6

    }
}
```

In-line details:

Wraps the constructor-injected publisher.

You need this empty constructor to satisfy the Contexts and Dependency Injection (CDI) for Java specification.

Subscribe to the delegate.

Wrap the delegate in a publisher that can handle multiple subscriptions.

The wrapping publisher forwards data from the delegate.

Unsubscribe from the reactive messaging-provided publisher.

In this example, MicroProfile Reactive Messaging is listening to the **`my-stream`** memory stream, so messages sent through the **`Emitter`** are received on this injected publisher. Note, though, that the following conditions must be true for this data exchange to succeed:

1. There must be an active subscription on the channel before you call **`Emitter.send()`**. In this example, notice that

the **`subscribeAndAllowMultipleSubscriptions()`** method called by the constructor ensures that there's an active subscription by the time the bean is available for user code calls.

2. You can have only one **`Subscription`** on the injected **`Publisher`**. If you want to expose the receiving publisher with a REST call, where each call to the **`poll()`** method results in a new subscription to the **`dest`** publisher, you have to implement your own publisher to broadcast data from the injected to each client.

13.5.3. The Apache Kafka user API

You can use the Apache Kafka user API to get more information about messages Kafka received, and to influence how Kafka handles messages. This API is stored in the **`io/smallrye/reactive/messaging/kafka/api`** package, and it consists of the following classes:

- **`IncomingKafkaRecordMetadata`**. This metadata contains the following information:
  - The Kafka record **`key`**, represented by a **`Message`**.
  - The Kafka **`topic`** and **`partition`** used for the **`Message`**, and the **`offset`** within those.
  - The **`Message timestamp`** and **`timestampType`**.
  - The **`Message headers`**. These are pieces of information that the application can attach on the producing side, and receive on the consuming side.
- **`OutgoingKafkaRecordMetadata`**. With this metadata, you can specify or override how Kafka handles messages. It contains the following information:
  - The **`key`**. which Kafka treats as the message key.
  - The **`topic`** you want Kafka to use.
  - The **`partition`**.
  - The **`timestamp`**, if you don't want the one that Kafka generates.
  - **`headers`**.
- **`KafkaMetadataUtil`** contains utility methods to write **`OutgoingKafkaRecordMetadata`** to a **`Message`**, and to read **`IncomingKafkaRecordMetadata`** from a **`Message`**.

> **Important**
>
> If you write **`OutgoingKafkaRecordMetadata`** to a **`Message`** sent to a channel that's not mapped to Kafka, the reactive messaging framework ignores it. Conversely, if you read **`IncomingKafkaRecordMetadata`** from a **`Message`** from a channel that's not mapped to Kafka, that message returns as **`null`**.

Example of how to write and read a message **key**

```
@Inject
@Channel("from-user")
Emitter<Integer> emitter;

@Incoming("from-user")
@Outgoing("to-kafka")
public Message<Integer> send(Message<Integer> msg) {
    // Set the key in the metadata
    OutgoingKafkaRecordMetadata<String> md =
            OutgoingKafkaRecordMetadata.<String>builder()
                .withKey("KEY-" + i)
                .build();
    // Note that Message is immutable so the copy returned
by this method
    // call is not the same as the parameter to the method
    return KafkaMetadataUtil.writeOutgoingKafkaMetadata(msg,
md);
}

@Incoming("from-kafka")
public CompletionStage<Void> receive(Message<Integer> msg) {
    IncomingKafkaRecordMetadata<String, Integer> metadata =

KafkaMetadataUtil.readIncomingKafkaMetadata(msg).get();

    // We can now read the Kafka record key
    String key = metadata.getKey();

    // When using the Message wrapper around the payload we
need to explicitly ack
    // them
    return msg.ack();
}
```

Example of Kafka mapping in a **`microprofile-config.properties`** file

```
kafka.bootstrap.servers=kafka:9092

mp.messaging.outgoing.to-kafka.connector=smallrye-kafka
mp.messaging.outgoing.to-kafka.topic=some-topic
mp.messaging.outgoing.to-
```

```
kafka.value.serializer=org.apache.kafka.common.serialization
.IntegerSerializer
mp.messaging.outgoing.to-
kafka.key.serializer=org.apache.kafka.common.serialization.S
tringSerializer

mp.messaging.incoming.from-kafka.connector=smallrye-kafka
mp.messaging.incoming.from-kafka.topic=some-topic
mp.messaging.incoming.from-
kafka.value.deserializer=org.apache.kafka.common.serializati
on.IntegerDeserializer
mp.messaging.incoming.from-
kafka.key.deserializer=org.apache.kafka.common.serialization
.StringDeserializer
```

Note

You must specify the **`key.serializer`** for the outgoing channel and the **`key.deserializer`** for the incoming channel.

13.5.4. Example MicroProfile Config properties file for the Kafka connector

This is an example of a simple **`microprofile-config.properties`** file for a Kafka connector. Its properties correspond to the properties in the example in "MicroProfile reactive messaging connectors for integrating with external messaging systems."

```
kafka.bootstrap.servers=kafka:9092

mp.messaging.outgoing.to.connector=smallrye-kafka
mp.messaging.outgoing.to.topic=my-topic
mp.messaging.outgoing.to.value.serializer=org.apache.kafka.c
ommon.serialization.IntegerSerializer

mp.messaging.incoming.from.connector=smallrye-kafka
mp.messaging.incoming.from.topic=my-topic
mp.messaging.incoming.from.value.deserializer=org.apache.kaf
ka.common.serialization.IntegerDeserializer
```

Table 13.6. Discussion of entries

Entry	Description

Entry	Description
`to`, **`from`**	These are "channels."
`send`, **`receive`**	These are "methods." Note that the **`to`** channel is on the **`send()`** method and the **`from`** channel is on the **`receive()`** method.
`kafka.bootstrap.servers=kafka:9092`	This specifies the URL of the Kafka broker that the application must connect to. You can also specify a URL at the channel level, like this: **`mp.messaging.outgoing.to.bootstrap.servers=kafka:9092`**
`mp.messaging.outgoing.to.connector=smallrye-kafka`	This indicates that you want the **`to`** channel to receive messages from Kafka. SmallRye reactive messaging is a framework for building applications. Note that the **`smallrye-kafka`** value is SmallRye reactive messaging-specific. If you're provisioning your own server using Galleon, you can enable the Kafka integration by including the **`microprofile-reactive-messaging-kafka`** Galleon layer.
`mp.messaging.outgoing.to.topic=my-topic`	This indicates that you want to send data to a Kafka topic called **`my-topic`**. A Kafka "topic" is a category or feed name that messages are stored on and published to. All Kafka messages are organized into topics. Producer applications write data *to* topics and consumer applications read data *from* topics.
`mp.messaging.outgoing.to.value.serializer=org.apache.kafka.common.serialization.IntegerSerializer`	This tells the connector to use **`IntegerSerializer`** to serialize the values that the **`send()`** method outputs when it writes to a topic. Kafka provides serializers for standard Java types. You can implement your own serializer by writing a class that implements **`org.apache.kafka.common.serialization.Serializer`**, and then include that class in your deployment.
`mp.messaging.incoming.from.connector=smallrye-kafka`	This indicates that you want to use the **`from`** channel to receive messages from Kafka. Again, the **`smallrye-kafka`** value is SmallRye reactive messaging-specific.
`mp.messaging.incoming.from.topic=my-topic`	This indicates that your connector should read data from the Kafka topic called **`my-topic`**.

Entry	Description
`mp.messaging.incoming.from.value.deserializer=org.apache.kafka.common.serialization.IntegerDeserializer`	This tells the connector to use **`IntegerDeserializer`** to deserialize the values from the topic before calling the **`receive()`** method. You can implement your own deserializer by writing a class that implements **`org.apache.kafka.common.serialization.Deserializer`**, and then include that class in your deployment.

Note

This list of properties is not comprehensive. See the SmallRye Reactive Messaging Apache Kafka documentation for more information.

Mandatory MicroProfile Reactive Messaging prefixes

The MicroProfile Reactive Messaging specification requires the following method property key prefixes for Kafka:

- **`mp.messaging.incoming.[channel-name].[attribute]=[value]`**
- **`mp.messaging.outgoing.[channel-name].[attribute]=[value]`**
- **`mp.messaging.connector.[connector-name].[attribute]=[value]`**

Note that **`channel-name`** is either the **`@Incoming.value()`** or the **`@Outgoing.value()`**.

Now consider the following method pair example:

```
@Outgoing("to")
public int send() {
    int i = // Randomly generated...
    return i;
}

@Incoming("from")
public void receive(int i) {
    // Process payload
}
```

In this method pair example, note the following required property prefixes:

- **`mp.messaging.incoming.from`**. This prefix selects the property as your configuration of the **`receive()`** method.
- **`mp.messaging.outgoing.to`**. This prefix selects the property as your configuration of the **`send()`** method.

13.5.5. Example MicroProfile Config properties file for the AMQP connector

This is an example of a simple **`microprofile-config.properties`** file for an Advanced Message Queuing Protocol (AMQP) connector. Its properties correspond to the properties in the example in MicroProfile reactive messaging connectors for integrating with external messaging systems.

```
amqp-host=localhost
amqp-port=5672
amqp-username=artemis
amqp-password=artemis

mp.messaging.outgoing.to.connector=smallrye-amqp
mp.messaging.outgoing.to.address=my-topic

mp.messaging.incoming.from.connector=smallrye-amqp
mp.messaging.incoming.from.address=my-topic
```

Table 13.7. Discussion of entries

Entry	Description
`to`, **`from`**	These are "channels."
`send`, **`receive`**	These are "methods." Note that the **`to`** channel is on the **`send()`** method and the **`from`** channel is on the **`receive()`** method.
`amqp-host=localhost`	This specifies the URL of the AMQP broker that the application must connect to. You can also specify a URL at the channel level, like this: **`mp.messaging.outgoing.to.host=localhost`**.The value defaults to **`localhost`** when no URL is specified.
`amqp-port=5672`	This specifies the port of the AMQP broker.
`mp.messaging.outgoing.to.connector=smallrye-amqp`	This indicates that you want the channel to send messages to AMQP. SmallRye reactive messaging is a framework for building applications. Note that the **`smallrye-amqp`** value is SmallRye reactive messaging specific. If you're provisioning your own server using Galleon, you can enable the AMQP integration by

Entry	Description
	including the **microprofile-reactive-messaging-amqp** Galleon layer.
mp.messaging.outgoing.to.address=my-topic	This indicates that you want to send data to an AMQP queue on the address **my-topic**. If you do not specify a value for **mp.messaging.outgoing.to.address**, the value will default to the channel, which in this example is "to".
mp.messaging.incoming.from.connector=smallrye-amqp	This indicates that you want to use the **from** channel to receive messages from the AMQP broker. Again, the **smallrye-amqp** value is SmallRye reactive messaging-specific.
mp.messaging.incoming.from.address=my-topic	This indicates that you want to read data from the AMQP queue **my-topic** on the **from** channel.

For a complete list of properties supported by the SmallRye Reactive Messaging's AMQP connector, see SmallRye Reactive Messaging AMQP Connector Configuration Reference.

Connecting to a secure AMQP broker

To connect with an AMQ broker secured with SSL/TLS and Simple Authentication and Security Layer (SASL), define the **client-ssl-context** to be used for the connection, in the **microprofile-config.properties** file. You can do this on connector level and also on channel level.

Example of connector level `client-ssl-context` definition

```
amqp-use-ssl=true
mp.messaging.connector.smallrye-
amqp.wildfly.elytron.ssl.context=exampleSSLContext
```

The attribute **mp.messaging.connector.smallrye-amqp.wildfly.elytron.ssl.context** is only required when you use self-signed certificates.

> **Important**
>
> Do not use self-signed certificates in a production environment. Use only the certificates signed by a certificate authority (CA).

You can also specify the **client-ssl-context** for a channel as follows:

Example of channel-level `client-ssl-context` definition

```
mp.messaging.incoming.from.wildfly.elytron.ssl.context=exampleSSLContext
```

In the example, the **`exampleSSLContext`** is associated only with the incoming channel **`from`**.

Table 13.8. Discussion of entries

Entry	Description
`amqp-use-ssl`	This specifies that we want to use a secure connection when connecting to the broker.
`mp.messaging.connector.smallrye-amqp.wildfly.elytron.ssl.context`	You do not need to specify this attribute if the AMQ broker is secured with a Certificate Authority (CA)-signed certificate. If you use a self-signed certificate, specify the **`SSLContext`** that is defined in the Elytron subsystem under **`/subsystem=elytron/client-ssl-context=*`** in the management model. **Important** Do not use self-signed certificates in a production environment. Use only the certificates signed by a certificate authority (CA). You can define **`client-ssl-context`** by using the following management CLI command: **Example** `/subsystem=elytron/client-ssl-context=exampleSSLContext:add(key-manager=exampleServerKeyManager,trust-manager=exampleTLSTrustManager)` For more information, see Configuring a trust store and a trust manager for client certificates, Configuring a server certificate for two-way SSL/TLS in the *Configuring SSL/TLS in JBoss EAP* guide.

13.6. OPENTELEMETRY REFERENCE

13.6.1. OpenTelemetry subsystem attributes

You can modify **`opentelemetry`** subsystem attributes to configure its behavior. The attributes are grouped by the aspect they configure: exporter, sampler, and span processor.

Table 13.9. Exporter attribute group

Attribute	Description	Default value
`endpoint`	The URL to which OpenTelemetry pushes traces. Set this to the URL where your exporter listens.	http://localhost:14250/
`exporter-type`	The exporter to which traces are sent. It can be one of the following: - **`jaeger`**. The exporter you use is Jaeger. - **`otlp`**. The exporter you use works with the OpenTelemetry protocol.	**`jaeger`**

Table 13.10. Sampler attribute group

Attribute	Description	Default value
`ratio`	The ratio of traces to export. The value must be between **`0.0`** and **`1.0`**. For example, to export one trace in every 100 traces created by an application, set the value to **`0.01`**. This attribute takes effect only if you set the attribute **`sampler-type`** as **`ratio`**.	

Table 13.11. Span processor attribute group

Attribute	Description	Default value
`batch-delay`	The interval in milliseconds between two consecutive exports by JBoss EAP. This attribute only takes effect if you set the attribute **`span-processor-type`** as **`batch`**.	**`5000`**
`export-timeout`	The maximum amount of time in milliseconds to allow for an export to complete before being cancelled.	**`30000`**
`max-export-batch-size`	The maximum number of traces that are published in each batch. This number should be should be lesser or equal to the value of **`max-queue-size`**. You can set this attribute only if you set the attribute **`span-processor-type`** as **`batch`**.	**`512`**
`max-queue-size`	The maximum number of traces to queue before exporting. If an application creates more traces, they are not recorded. This attribute only takes effect if you set the attribute **`span-processor-type`** as **`batch`**.	**`2048`**

Attribute	Description	Default value

`span-processor-type`	The type of span processor to use. The value can be one of the following:
	- **`batch`**: JBoss EAP exports traces in batches that are defined using the following attributes:
	  - **`batch-delay`**
	  - **`max-export-batch-size`**
	  - **`max-queue-size`**
	- **`simple`**: JBoss EAP exports traces are as soon as they finish.
		`batch`

Additional resources

- OpenTelemetry in JBoss EAP

LEGAL NOTICE