Red Hat JBoss Enterprise Application Platform 7.4

使用 JBoss EAP XP 4.0.0

用于 JBoss EAP XP 4.0.0

Last Updated: 2024-03-21

Red Hat JBoss Enterprise Application Platform 7.4 使用 JBoss EAP XP 4.0.0

用于 JBoss EAP XP 4.0.0

法律通告

摘要

本文档提供有关在 JBoss EAP XP 4.0.0 中使用 MicroProfile 的一般信息。

目录

使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。由于此项工作十分艰巨，这些更改将在即将推出的几个发行版本中逐步实施。详情请查看 CTO Chris Wright 的信息。

提供有关 JBOSS EAP 文档的反馈

要报告错误或改进文档，请登录到 Red Hat JIRA 帐户并提交问题。如果您没有 Red Hat Jira 帐户，则会提示您创建一个帐户。

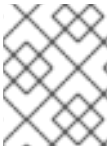
流程

1. 单击以下链接 **以创建 ticket**。
2. 请包含 **文档 URL、章节编号 并描述问题**。
3. 在 **Summary** 中输入问题的简短描述。
4. 在 **Description** 中提供问题或功能增强的详细描述。包括一个指向文档中问题的 URL。
5. 点 **Submit** 创建问题，并将问题路由到适当的文档团队。

第 1 章 JBOSS EAP XP 用于最新的 MICROPROFILE 功能

1.1. 关于 JBOSS EAP XP

MicroProfile 扩展包(JBoss EAP XP)作为补丁流提供，该流使用 JBoss EAP XP 管理器提供。



注意

JBoss EAP XP 遵循单独的支持和生命周期政策。详情请查看 JBoss Enterprise Application Platform 扩展软件包支持和生命周期政策页。

JBoss EAP XP 补丁提供以下 MicroProfile 4.1 组件：

- MicroProfile Config
- MicroProfile Fault Tolerance
- MicroProfile Health
- MicroProfile JWT
- MicroProfile Metrics
- MicroProfile OpenAPI
- MicroProfile OpenTracing
- MicroProfile REST 客户端
- MicroProfile Reactive Messaging

注意

MicroProfile Reactive Messaging 子系统支持 Red Hat AMQ Streams。此功能实施 MicroProfile Reactive Messaging 2.0.1 API，红帽提供该功能作为 JBoss EAP XP 4.0.0 的技术预览。

红帽在 JBoss EAP 中测试了 Red Hat AMQ Streams 2021.Q4。但是，检查 Red Hat JBoss Enterprise Application Platform 支持的配置页面，以了解有关在 JBoss EAP XP 4.0.0 中测试的最新 Red Hat AMQ Streams 版本的信息。

技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。有关红帽技术预览功能支持范围的更多信息，请参阅技术预览功能支持范围。

1.2. JBOSS EAP XP 安装

安装 JBoss EAP XP 时，请确保 JBoss EAP XP 补丁与您的 JBoss EAP 版本兼容。JBoss EAP XP 4.0.x 补丁与 JBoss EAP 7.4 版本兼容。

注意

您可以通过 XP 管理器和 EAP 存档或使用 JBoss EAP XP OpenShift 容器镜像安装 JBoss EAP XP。您不能在 EAP RPM 之上安装 JBoss EAP XP。

其它资源

- 有关在最新的 JBoss EAP 版本上安装最新的 JBoss EAP XP 补丁的更多信息，请参阅在 JBoss EAP 7.4.x 上安装 JBoss EAP XP 4.0.0。

1.3. JBOSS EAP XP MANAGER

JBoss EAP XP 管理器是一个可执行的 **jar** 文件，您可以从 **产品** 下载页面下载。使用 JBoss EAP XP 管理器从 JBoss EAP XP 补丁流应用 JBoss EAP XP 补丁。补丁包含 MicroProfile 4.1 实施，以及这些 MicroProfile 4.1 实施的 bug 修复。

注意

您不能使用管理控制台管理 JBoss EAP XP 补丁。

如果您运行不带任何参数的 JBoss EAP XP 管理器，或者使用 **help** 命令，获得所有可用命令的列表，其中包含它们的作用信息。

使用 **help** 命令运行管理器，以获取有关可用参数的更多信息。

注意

大多数 JBoss EAP XP 管理器命令使用 **--jboss-home** 参数指向 JBoss EAP XP 服务器来管理 JBoss EAP XP 补丁流。如果要省略它，请指定 **JBOSS_HOME** 环境变量中的服务器路径。**--jboss-home** 优先于环境变量。

1.4. JBOSS EAP XP MANAGER 4.0 命令

JBoss EAP XP 管理器 4.0 提供了不同的命令来管理 JBoss EAP XP 补丁流。

提供以下命令：

patch-apply

使用此命令将补丁应用到您的 JBoss EAP 安装。
patch-apply 命令与 **patch apply** 管理 CLI 命令类似。**patch-apply** 命令只接受使用工具应用补丁所需的参数。它将默认值用于其他 **补丁应用** 管理 CLI 命令参数。

您可以使用 **patch-apply** 命令将补丁应用到服务器上启用的任何补丁流。您还可以使用 命令应用基本服务器补丁和 XP 补丁。

使用 patch-apply 命令的示例：

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

应用 XP 补丁时，JBoss EAP XP 管理器 4.0 会执行验证，以防止补丁和补丁流不匹配。以下示例演示了不正确的组合：

- 在设置了 XP 4.0 补丁流的服务器上安装 JBoss EAP XP 3.0 补丁会导致以下错误：

```
java.lang.IllegalStateException: The JBoss EAP XP patch stream in the patch 'jboss-eap-xp-3.0' does not match the currently enabled JBoss EAP XP patch stream [jboss-eap-xp-4.0]
 at org.jboss.eap.util.xp.patch.stream.manager.ManagerPatchApplyAction.doExecute(ManagerPatchApplyAction.java:33)
 at org.jboss.eap.util.xp.patch.stream.manager.ManagerAction.execute(ManagerAction.java:40)
 at org.jboss.eap.util.xp.patch.stream.manager.ManagerMain.main(ManagerMain.java:50)
```

- 尝试在未为 JBoss EAP XP 4.0.0 补丁流设置的服务器上安装 JBoss EAP XP 4.0.0 补丁程序会导致以下错误：

```
java.lang.IllegalStateException: You are attempting to install a patch for the 'jboss-eap-xp-4.0' JBoss EAP XP Patch Stream. However this patch stream is not yet set up in the JBoss EAP server. Run the 'setup' command to enable the patch stream.
 at org.jboss.eap.util.xp.patch.stream.manager.ManagerPatchApplyAction.doExecute(ManagerPatchApplyAction.java:29)
 at org.jboss.eap.util.xp.patch.stream.manager.ManagerAction.execute(ManagerAction.java:40)
 at org.jboss.eap.util.xp.patch.stream.manager.ManagerMain.main(ManagerMain.java:50)
```

在这两种情况下，不会对服务器进行任何更改。

remove

使用此命令从 JBoss EAP 服务器中删除 JBoss EAP XP 补丁流设置。

使用 remove 命令的示例

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

setup

使用此命令为 JBoss EAP XP 补丁流设置干净的 JBoss EAP 服务器。
使用 **setup** 命令时，JBoss EAP XP manager 执行以下操作：

- 启用 JBoss EAP XP 4.0.0 补丁流。
- 应用使用 **--base-patch** 和 **--xp-patch** 属性指定的补丁。
- 将 **standalone-microprofile.xml** 和 **standalone-microprofile-ha.xml** 配置文件复制到服务器配置目录中。
  如果已安装较旧的配置文件，新文件将保存为目标配置目录中的时间戳副本，如 **standalone-microprofile-yyyyMMdd-HHmmss.xml**。

  您可以使用 **--jboss-config-directory** 参数设置目标目录。

使用 setup 命令的示例

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

status

使用此命令查找 JBoss EAP XP 服务器的当前状态。status 命令返回以下信息：

- JBoss EAP XP 流的状态。
- 由于当前状态处于当前状态，任何支持策略都会改变。
- JBoss EAP XP 的主版本。
- 启用补丁流及其累积补丁 ID。
- 可用的 JBoss EAP XP manager 命令更改状态。

使用 status 命令的示例

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

Upgrade （升级）

使用此命令将旧的 JBoss EAP XP 补丁流升级到 JBoss EAP 服务器的最新补丁流。
使用 **upgrade** 命令时，JBoss EAP XP manager 会执行以下操作：

- 创建文件的备份，启用服务器中的旧补丁流。
- 启用 JBoss EAP XP 4.0 补丁流。
- 应用使用 **--base-patch** 和 **--xp-patch** 属性指定的补丁。
- 将 **standalone-microprofile.xml** 和 **standalone-microprofile-ha.xml** 配置文件复制到服务器配置目录中。如果已安装较旧的配置文件，新文件将保存为目标配置目录中的时间戳副本，如 **standalone-microprofile-yyyyMMdd-HHmmss.xml**。
- 如果出现错误，JBoss EAP XP Manager 会尝试从创建的备份中恢复以前的补丁流。
  您可以使用 **--jboss-config-directory** 参数设置目标目录

使用 upgrade 命令的示例：

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

1.5. 在 JBOSS EAP 7.4.X 上安装 JBOSS EAP XP 4.0.0

在 JBoss EAP 7.4 基础服务器上安装 JBoss EAP XP 4.0.0。

使用 JBoss EAP XP 管理器 4.0.0 管理 JBoss EAP XP 4.0.0 补丁流。

注意

JBoss EAP XP 4.0.0 使用 JBoss EAP 7.4.x 认证。

先决条件

- 您已从 **产品** 下载页面下载了以下文件：
  - **jboss-eap-xp-4.0.0-manager.jar** 文件(JBoss EAP XP manager 4.0)
  - JBoss EAP 7.4 服务器存档文件
  - JBoss EAP XP 4.0.0 补丁

流程

1. 将下载的 JBoss EAP 7.4 服务器存档文件提取到 JBoss EAP 安装的路径。
2. 使用以下命令设置 JBoss EAP XP 管理器 4.0.0，以管理 JBoss EAP XP 4.0 补丁流：

```
$ java -jar jboss-eap-xp-manager.jar setup --jboss-home=<path_to_eap>
```

注意

您可以同时应用 JBoss EAP XP 4.0.0 补丁。使用 **--xp-patch** 参数包括到 JBoss EAP XP 4.0.0 补丁的路径。

Example:

```
$ java -jar jboss-eap-xp-manager.jar setup --jboss-home=<path_to_eap> --xp-patch=<path_to_patch>jboss-eap-xp-4.0.0-patch.zip
```

服务器现在已准备好管理 JBoss EAP XP 4.0.0 补丁流。

3. 可选： 如果您还没有使用 **--xp-patch** 参数将 JBoss EAP XP 4.0.0 补丁应用到 JBoss EAP 服务器，请使用 JBoss EAP XP manager 4.0.0 **patch-apply** 命令应用 JBoss EAP XP 4.0.0 补丁：

```
$ java -jar jboss-eap-xp-manager.jar patch-apply --jboss-home=<path_to_eap> --patch=<path_to_patch>jboss-eap-xp-4.0.0-patch.zip
```

patch-apply 命令与 **patch apply** 管理 CLI 命令类似。您还可以使用 **patch apply** 管理 CLI 命令应用补丁。

现在，JBoss EAP 服务器已准备好管理 JBoss EAP XP 4.0.0 补丁流，因为您使用 JBoss EAP XP 4.0.0 补丁对 JBoss EAP 服务器进行补丁。

其它资源

- JBoss EAP XP manager 4.0 命令

1.6. 卸载 JBOSS EAP XP

卸载 JBoss EAP XP 会删除与启用 JBoss EAP XP 4.0.0 补丁流和 MicroProfile 4.1 功能相关的所有文件。卸载过程不会影响基本服务器补丁流或功能中的任何内容。

注意

卸载过程不会删除任何配置文件，包括在启用 JBoss EAP XP 补丁流时添加到 JBoss EAP XP 补丁程序中的配置文件。

流程

- 运行以下命令卸载 JBoss EAP XP 4.0.0：

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

要再次安装 MicroProfile 4.1 功能，请再次运行 **setup** 命令以启用补丁流，然后应用 JBoss EAP XP 补丁来添加 MicroProfile 4.1 模块。

1.7. 查看 JBOSS EAP XP 的状态

您可以使用 **status** 命令查看以下信息：

- JBoss EAP XP 流的状态。
- 由于当前状态处于当前状态，任何支持策略都会改变。
- JBoss EAP XP 的主版本。
- 启用补丁流及其累积补丁 ID。
- 可用的 JBoss EAP XP manager 命令更改状态。

JBoss EAP XP 可以处于以下状态之一：

未设置

JBoss EAP 非常干净，且没有设置 JBoss EAP XP。

设置

JBoss EAP 已设置 JBoss EAP XP。XP 补丁流的版本不会显示，因为用户可以使用 CLI 来确定它。

Inconsistent

与 JBoss EAP XP 相关的文件处于不一致状态。这是错误条件，不应正常发生。如果您遇到这个错误，请按照 卸载 JBoss EAP XP 主题中所述删除 JBoss EAP XP 管理器，并使用 **setup** 命令再次安装 JBoss EAP XP。

流程

- 运行以下命令，查看 JBoss EAP XP 的状态：

```
$ java -jar jboss-eap-xp-manager.jar status --jboss-home=<path_to_eap>
```

其它资源

- 卸载 JBoss EAP XP
- 在 JBoss EAP 7.4.x 上安装 JBoss EAP XP 4.0.0

1.8. 回滚 JBOSS EAP XP 和 JBOSS EAP 7.4.X 基础补丁

您可以使用管理 CLI 回滚之前应用的 JBoss EAP XP 补丁或 JBoss EAP 7.4.x 基础补丁。

其他资源

- 有关回滚 JBoss EAP XP 补丁或 JBoss EAP 7.4.x 基础补丁的更多信息，请参阅使用 管理 CLI 回滚补丁。

第 2 章 了解 MICROPROFILE

2.1. MICROPROFILE CONFIG

2.1.1. JBoss EAP 中的 MicroProfile 配置

配置数据可以动态更改，应用程序需要在不重新启动服务器的情况下访问最新的配置信息。

MicroProfile 配置提供可移植外部化配置数据。这意味着，您可以将应用程序和微服务配置为在多个环境中运行，而无需修改或重新打包。

MicroProfile 配置功能在 JBoss EAP 中使用 SmallRye Config 组件实施，它由 **microprofile-config-smallrye** 子系统提供。

注意

MicroProfile Config 仅在 JBoss EAP XP 中受到支持。JBoss EAP 不支持它。

重要

如果要添加自己的配置实现，则需要使用最新版本的 Config 接口中的方法。

其它资源

- MicroProfile Config
- 小配置
- 配置实现

2.1.2. MicroProfile Config 支持的 MicroProfile Config 源

MicroProfile 配置属性可以来自不同位置，可以采用不同的格式。这些属性由 ConfigSources 提供。ConfigSources 是 **org.eclipse.microprofile.config.spi.ConfigSource** 接口的实现。

MicroProfile Config 规范提供以下用于检索配置值的默认 **ConfigSource** 实施：

- **System.getProperties()**.
- **system.getenv** （）.
- 类路径上的所有 **META-INF/microprofile-config.properties** 文件。

microprofile-config-smallrye 子系统支持用于检索配置值的其他类型的 **ConfigSource** 资源。您还可以从以下资源检索配置值：

- **microprofile-config-smallrye/config-source** 管理资源中的属性
- 目录中的文件
- **ConfigSource** 类
- **ConfigSourceProvider** 类

其它资源

- **org.jboss.resteasy.microprofile.config.BaseServletConfigSource**

2.2. MICROPROFILE FAULT TOLERANCE

2.2.1. 关于 MicroProfile 容错规范

MicroProfile Fault Tolerance 规范定义了处理分布式微服务中固有的错误的策略。

MicroProfile Fault Tolerance 规范定义了以下策略来处理错误：

Timeout（超时）
定义执行完成的时间长度。定义超时可防止无限期等待执行。

Retry
定义重试失败的执行的条件。

fallback
在执行失败时提供替代方案。

CircuitBreaker
定义在临时停止前失败的执行尝试次数。您可以在恢复执行前定义延迟的长度。

bulkhead
在系统的一部分隔离故障，以便系统的其余部分仍然可以正常工作。

asynchronous
在单独的线程中执行客户端请求。

其它资源

- MicroProfile Fault Tolerance 规格

2.2.2. JBoss EAP 中的 MicroProfile Fault Tolerance

microprofile-fault-tolerance-smallrye 子系统为 JBoss EAP 中的 MicroProfile Fault Tolerance 提供支持。子系统仅在 JBoss EAP XP 流中可用。

microprofile-fault-tolerance-smallrye 子系统为拦截器绑定提供以下注释：

- **@timeout**
- **@retry**
- **@Fallback**
- **@CircuitBreaker**
- **@Bulkhead**
- **@Asynchronous**

您可以在类级别或方法级别上绑定这些注解。绑定到某个类的注解适用于该类的所有业务方法。

以下规则适用于绑定拦截器：

- 如果组件类声明或继承类级拦截器绑定，则有以下限制：
  - 不得声明最终的类。
  - 类不得包含任何静态、私有或最终方法。
- 如果一个非静态的、组件类的非私有方法声明方法级别拦截器绑定，则方法和组件类都不能声明最终声明。

容错操作有以下限制：

- 容错拦截器绑定必须应用到 bean 类或 bean 类方法。
- 调用时，调用必须是 Jakarta 上下文和依赖注入规范中定义的业务方法调用。
- 如果这两个条件都满足，则操作不被视为容错：
  - 该方法本身不绑定到任何容错拦截器。
  - 包含方法的类不绑定到任何容错拦截器。

除了 MicroProfile Fault Tolerance 提供的配置选项外，**microprofile-fault-tolerance-smallrye** 子系统还提供以下配置选项：

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

其它资源

- MicroProfile 容错规范
- 小型容错项目

2.3. MICROPROFILE HEALTH

2.3.1. JBoss EAP 中的 MicroProfile Health

JBoss EAP 包含 SmallRye Health 组件，您可以使用它来确定 JBoss EAP 实例是否按预期响应。此功能默认为启用。

MicroProfile Health 仅在将 JBoss EAP 作为单机服务器运行时可用。

MicroProfile 健康规范定义以下健康检查：

就绪
: 决定应用程序是否准备好处理请求。注释 **@Readiness** 提供此健康检查。

Liveness
: 确定应用程序是否正在运行。注释 **@Liveness** 提供此健康检查。

启动
: 确定应用程序是否已启动。注释 **@Startup** 提供此健康检查。

MicroProfile Health 3.0 中删除了 **@Health** 注释。

MicroProfile Health 3.1 包含一个新的 **启动** 健康检查探测。

有关 MicroProfile Health 3.1 更改的更多信息，请参阅 MicroProfile Health 3.1 发行注记。

重要

:empty-readiness-checks-status,**:empty-liveness-checks-status**, 和 **:empty-startup-checks-status** 管理属性指定没有 **就绪度**、**存活度** 或 **启动探测** 的全局状态。

其它资源

- 没有定义探测时的全局状态
- smallrye Health
- MicroProfile Health
- 自定义健康检查示例

2.4. MICROPROFILE JWT

2.4.1. JBoss EAP 中的 MicroProfile JWT 集成

子系统 **microprofile-jwt-smallrye** 在 JBoss EAP 中提供 MicroProfile JWT 集成。

以下功能由 **microprofile-jwt-smallrye** 子系统提供：

- 检测使用 MicroProfile JWT 安全性的部署。
- 激活对 MicroProfile JWT 的支持.

该子系统不包含可配置的属性或资源。

除了 **microprofile-jwt-smallrye** 子系统外，**org.eclipse.microprofile.jwt.auth.api** 模块在 JBoss EAP 中提供 MicroProfile JWT 集成。

其它资源

- SmallRye JWT

2.4.2. 传统部署和 MicroProfile JWT 部署之间的区别

MicroProfile JWT 部署不依赖于传统 JBoss EAP 部署等管理的 SecurityDomain 资源。相反，在 MicroProfile JWT 部署中创建并使用了一个虚拟 SecurityDomain。

由于 MicroProfile JWT 部署完全在 MicroProfile 配置属性和 **microprofile-jwt-smallrye** 子系统内配置，因此虚拟 SecurityDomain 不需要任何其他受管配置进行部署。

2.4.3. JBoss EAP 中的 MicroProfile JWT 激活

根据应用中存在 **auth-method**，为应用激活 MicroProfile JWT。

以以下方式为应用程序激活 MicroProfile JWT 集成：

- 作为部署过程的一部分，JBoss EAP 会扫描应用存档是否存在 **auth-method**。

- 如果存在 **auth-method** 并定义为 **MP-JWT**，则会激活 MicroProfile JWT 集成。

auth-method 可以在以下一个或多个文件中指定：

- 包含扩展 **javax.ws.rs.core.Application** 的类的文件，它标上 **@LoginConfig**
- **web.xml** 配置文件

如果在类中使用注解和 web.xml 配置文件中定义了 **auth-method**，则会使用 **web.xml** 配置文件中的定义。

2.4.4. JBoss EAP 中的 MicroProfile JWT 的限制

JBoss EAP 中的 MicroProfile JWT 实施存在一些限制。

JBoss EAP 中存在以下 MicroProfile JWT 实施限制：

- MicroProfile JWT 实施仅解析 **mp.jwt.verify.publickey** 属性中提供的 JSON Web 密钥集 (JWKS)的第一个密钥。因此，如果令牌声明由第二个密钥或第二个密钥后的任何密钥签名，令牌将无法验证，并且包含令牌的请求不会被授权。
- 不支持 JWKS 的 Base64 编码。

在这两种情况下，都可引用明文 JWKS，而不使用 **mp.jwt.publickey.location** 配置属性。

2.5. MICROPROFILE METRICS

2.5.1. JBoss EAP 中的 MicroProfile 指标

JBoss EAP 包括 SmallRye Metrics 组件。SmallRye Metrics 组件使用 **microprofile-metrics-smallrye** 子系统提供 MicroProfile 指标功能。

microprofile-metrics-smallrye 子系统为 JBoss EAP 实例提供监控数据。子系统默认为启用。

重要

microprofile-metrics-smallrye 子系统仅在独立配置中启用。

其它资源

- 小指标
- MicroProfile Metrics

2.6. MICROPROFILE OPENAPI

2.6.1. JBoss EAP 中的 MicroProfile OpenAPI

MicroProfile OpenAPI 使用 **microprofile-openapi-smallrye** 子系统在 JBoss EAP 中集成。

MicroProfile OpenAPI 规范定义提供 OpenAPI 3.0 文档的 HTTP 端点。OpenAPI 3.0 文档描述了主机的 REST 服务。OpenAPI 端点使用配置的路径注册，如 http://localhost:8080/openapi，到与部署关联的主机根目录。

注意

目前，虚拟主机的 OpenAPI 端点只能记录单个部署。要将 OpenAPI 与同一虚拟主机上不同上下文路径注册的多个部署搭配使用，每个部署都必须使用不同的端点路径。

OpenAPI 端点默认返回 YAML 文档。您还可以使用 Accept HTTP 标头或格式查询参数请求 JSON 文档。

如果给定应用的 Undertow 服务器或主机定义了 HTTPS 侦听器，则 OpenAPI 文档也可使用 HTTPS。例如，HTTPS 的端点是 https://localhost:8443/openapi。

2.7. MICROPROFILE OPENTRACING

2.7.1. MicroProfile OpenTracing

在跨服务界限跟踪请求的能力非常重要，特别是在请求可以在其生命周期内通过多个服务的微服务环境中。

MicroProfile OpenTracing 规范定义了用于访问 CDI-bean 应用内 OpenTracing 兼容 **Tracer** 接口的行为和 API。**Tracer** 接口会自动跟踪 JAX-RS 应用。

此行为指定如何为传入和传出请求自动创建 OpenTracing Span。API 定义了如何为给定端点显式禁用或启用追踪。

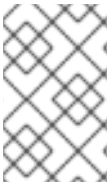
其它资源

- 有关 MicroProfile OpenTracing 规范的更多信息，请参阅 MicroProfile OpenTracing 文档。
- 有关 **Tracer** 接口的更多信息，请参阅 **Tracer** javadoc。

2.7.2. JBoss EAP 中的 MicroProfile OpenTracing

您可以使用 **microprofile-opentracing-smallrye** 子系统来配置 Jakarta EE 应用的分布式追踪。此子系统使用 SmallRye OpenTracing 组件为 JBoss EAP 提供 MicroProfile OpenTracing 功能。

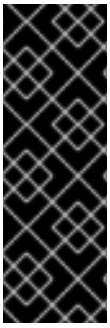
MicroProfile OpenTracing 2.0 支持追踪应用的请求。您可以配置默认的 Jaeger Java Client tracer，以及一组用于 Jakarta EE 常用的组件的检测库，通过管理 CLI 或管理控制台使用 JBoss EAP 管理 API。



注意

部署到 JBoss EAP 服务器的每个 WAR 自动具有自己的 **Tracer** 实例。EAR 中的每个 WAR 被视为单独的 WAR，各自具有自己的 **Tracer** 实例。默认情况下，与 Jaeger Client 一起使用的服务名称从部署的名称派生，通常是 WAR 文件名。

在 **microprofile-opentracing-smallrye** 子系统内，您可以通过设置系统属性或环境变量来配置 Jaeger Java 客户端。



重要

使用系统属性和环境变量配置 Jaeger Client tracer 作为技术预览提供。与 Jaeger Client tracer 集成的系统属性和环境变量可能会改变，并在以后的版本中相互不兼容。

技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。有关红帽技术预览功能支持范围的更多信息，请参阅技术预览功能支持范围。

注意

默认情况下，Java 的 Jaeger Client 的 probabilistic 抽样策略被设置为 **0.001**，这意味着在大约一个千个 trace 中被抽样。要抽样每个请求，请将系统属性 **JAEGER_SAMPLER_TYPE** 设置为 **const**，**JAEGER_SAMPLER_PARAM** 设置为 **1**。

其它资源

- 有关 SmallRye OpenTracing 功能的更多信息，请参阅 SmallRye OpenTracing 组件。
- 有关默认 tracer 的更多信息，请参阅 Jaeger Java Client。
- 有关 **Tracer** 接口的更多信息，请参阅 **Tracer** javadoc。
- 有关覆盖默认 tracer 和追踪 Jakarta 上下文和依赖注入 Bean 的更多信息，请参阅 *开发指南中的使用 Eclipse MicroProfile OpenTracing 追踪请求*。
- 有关配置 Jaeger Client 的更多信息，请参阅 Jaeger 文档。
- 有关有效系统属性的更多信息，请参阅 Jaeger 文档中的通过 环境配置。

2.8. MICROPROFILE REST 客户端

2.8.1. MicroProfile REST 客户端

JBoss EAP XP 4.0.0 支持基于 Jakarta RESTful Web Services 2.1.6 客户端 API 构建的 MicroProfile REST 客户端 2.0，以提供通过 HTTP 调用 RESTful 服务的类型安全方法。MicroProfile Type Safe REST 客户端被定义为 Java 接口。借助 MicroProfile REST 客户端，您可以使用可执行代码编写客户端应用。

使用 MicroProfile REST 客户端提供以下功能：

- 直观的语法
- 供应商编程注册
- 供应商声明注册
- 标头的声明规格
- **ResponseExceptionMapper**
- Jakarta 上下文和依赖注入集成
- 访问服务器事件(SSE)

其他资源

- MicroProfile REST 客户端和 Jakarta RESTful Web Services 语法之间的比较
- 在 MicroProfile REST 客户端中编程注册提供程序
- MicroProfile REST 客户端中的提供程序声明注册
- MicroProfile REST 客户端中的标头声明规格
- MicroProfile REST 客户端中的 ResponseExceptionMapper
- 使用 MicroProfile REST 客户端进行上下文依赖项注入

2.8.2. resteasy.original.webapplicationexception.behavior MicroProfile Config 属性

MicroProfile Config 是可供开发人员用来将应用和微服务配置为在多个环境中运行的规范名称，而无需修改或重新打包这些应用程序。在以前的版本中，MicroProfile Config 作为技术预览提供 JBoss EAP，但自此后已被删除。MicroProfile 配置现在仅适用于 JBoss EAP XP。

定义 resteasy.original.webapplicationexception.behavior MicroProfile Config 属性
您可以将 **resteasy.original.webapplicationexception.behavior** 参数设置为 **web.xml** servlet 属性或系统属性。以下是 **web.xml** 中一个 such servlet 属性的示例：

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

您还可以使用 MicroProfile Config 来配置任何其他 RESTEasy 属性。

其他资源

- 如需有关 JBoss EAP XP 上的 MicroProfile 配置的更多信息，请参阅了解 *MicroProfile*。
- 有关 MicroProfile REST 客户端的更多信息，请参阅 MicroProfile REST 客户端。
- 有关 RESTEasy 的更多信息，请参阅 Jakarta RESTful Web Services Request Processing。

2.9. MICROPROFILE REACTIVE MESSAGING

2.9.1. MicroProfile 被动消息传递

当您升级到 JBoss EAP XP 4.0.0 时，您可以启用最新版本的 MicroProfile Reactive 消息传递，其中包括被动消息传递扩展和子系统。

"主动流"是事件数据的成功，以及处理协议和标准，并在没有缓冲区的异步边界（如调度程序）间推送。"事件"可能是调度，并在 hwlatdetect 应用中重复温度检查，例如：被动流的主要优点是您的各种应用程序和实现的无缝互操作性。

被动消息传递为构建事件驱动的、数据流和事件源应用程序提供了一个框架。被动消息传递导致事件数据的持续和平稳交换（从被动流到另一个应用程序）。您可以使用 MicroProfile Reactive Messaging 通过被动流进行异步消息传递，以便应用程序可以与其他流（如 Apache Kafka）进行交互。

在将 MicroProfile Reactive Messaging 实例升级到最新版本后，您可以执行以下操作：

- 为 Apache Kafka 数据流平台置备带有 MicroProfile Reactive Messaging 的服务器。
- 与被动消息传递 in-memory 交互，并通过最新的被动消息传递 API 支持 Apache Kafka 主题。
- 使用 MicroProfile 指标查找在给定频道上流出多少消息。

其他资源

- 有关 Apache Kafka 的更多信息，请参阅什么是 Apache Kafka？

2.9.2. MicroProfile 被动消息传递连接器

您可以使用连接器将 MicroProfile 被动消息传递与多个外部消息传递系统集成。用于 JBoss EAP 的 MicroProfile 附带 Apache Kafka 连接器。使用 Eclipse MicroProfile Config 规范配置您的连接器。

Apache Kafka 连接器并融合了层
MicroProfile Reactive Messaging 包括 Kafka 连接器，您可以使用 MicroProfile 配置进行配置。Kafka 连接器融合了 **microprofile-reactive-messaging-kafka** 和 **microprofile-reactive-messaging** Galleon 层。**microprofile-reactive-messaging** 层提供核心 MicroProfile 主动消息功能。

表 2.1. 主动消息传递和 Apache Kafka 连接器 Galleon 层

layer	定义
microprofile-reactive-streams-operators	• 提供 MicroProfile Reactive Streams Operator API 和支持实施模块。 • 包含具有 SmallRye 扩展和子系统的 MicroProfile Reactive Streams Operator。 • 取决于 **cdi** 层。 ◦ **CDI** 代表 Jakarta 上下文和依赖注入；提供添加 **@Inject** 功能的子系统。
microprofile-reactive-messaging	• 提供 MicroProfile 主动消息传递 API 和支持实施模块。 • 包含具有 SmallRye 扩展和子系统的 MicroProfile。 • 依赖于 **microprofile-config** 和 **microprofile-reactive-streams-operators** 层。
microprofile-reactive-messaging-kafka	• 提供 Kafka 连接器模块，使 MicroProfile 主动消息传递能够与 Kafka 交互。 • 依赖于 **microprofile-reactive-messaging** 层。

2.9.3. Apache Kafka 事件流平台

Apache Kafka 是一个开源分布式事件（数据）流平台，可以实时发布、订阅、存储和处理记录流。它处理来自多个源的事件流，并将其提供给多个消费者，将大量数据从 A 移到 Z 和其他地方，并同时将其处理。MicroProfile Reactive 消息传递使用 Apache Kafka 在几微秒内提供这些事件记录，将其安全存储在分布式、容错集群中，同时将其在任何团队定义区域或地理区域中可用。

其他资源

- 什么是 Apache Kafka？
- Red Hat OpenShift Streams for Apache Kafka
- Red Hat AMQ

第 3 章 在 JBOSS EAP 中管理 MICROPROFILE

3.1. MICROPROFILE OPENTRACING 管理

重要

如果您看到为 REST 调用导出的重复 trace，请禁用 **microprofile-opentracing-smallrye** 子系统。有关禁用 **microprofile-opentracing-smallrye** 的详情请参考 删除 microprofile-opentracing-smallrye 子系统。

3.1.1. 启用 MicroProfile Open Tracing

使用以下管理 CLI 命令，通过将子系统添加到服务器配置，为服务器实例全局启用 MicroProfile Open Tracing 功能。

流程

1. 使用以下管理命令启用 **microprofile-opentracing-smallrye** 子系统：

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

2. 重新加载服务器以使更改生效。

```
reload
```

3.1.2. 删除 microprofile-opentracing-smallrye 子系统

microprofile-opentracing-smallrye 子系统包含在默认的 JBoss EAP 7.4 配置中。此子系统为 JBoss EAP 7.4 提供 MicroProfile OpenTracing 功能。如果您在启用 MicroProfile OpenTracing 时遇到系统内存或性能下降，您可能想禁用 **microprofile-opentracing-smallrye** 子系统。

您可以在管理 CLI 中使用 **remove** 操作，为给定服务器全局禁用 MicroProfile OpenTracing 功能。

流程

1. 删除 **microprofile-opentracing-smallrye** 子系统。

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

2. 重新加载服务器以使更改生效。

```
reload
```

3.1.3. 安装 Jaeger

使用 **docker** 安装 Jaeger。

先决条件

- **已安装** Docker。

流程

1. 在 CLI 中运行以下命令来使用 **docker** 安装 Jaeger：

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

3.2. MICROPROFILE 配置配置

3.2.1. 在 ConfigSource 管理资源中添加属性

您可以直接将属性存储在 **config-source** 子系统中，作为管理资源。

流程

- 创建 ConfigSource 并添加属性：

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

3.2.2. 将目录配置为 ConfigSources

当属性作为文件存储在目录中时，file-name 是属性的名称，文件内容是 属性的值。

流程

1. 创建一个要存储文件的目录：

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

2. 进入该目录：

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

3. 创建 **文件名** 来存储属性名为 的值：

```
$ touch name
```

4. **将属性值添加到文件中：**

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

5. **创建一个 ConfigSource，其中文件名是属性，文件内容为属性值：**

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

这会生成以下 XML 配置：

```
<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. 从 ConfigSource 类获取 ConfigSource

您可以创建并配置自定义 org.eclipse.microprofile.config.spi.ConfigSource 实施类，以便为配置值提供源。

流程

- 以下管理 CLI 命令为名为 org.example.MyConfigSource 的实施类创建一个 ConfigSource，它由名为 org.example 的 JBoss 模块提供。

  如果要使用 org.example 模块中的 ConfigSource，请将 < module name="org.eclipse.microprofile.config.api"/> 依赖项添加到 path/to/org/example/main/module.xml 文件。

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

  此命令生成 microprofile-config-smallrye 子系统的以下 XML 配置：

```
<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>
```

自定义 org.eclipse.microprofile.config.spi.ConfigSource 实施类提供的属性可供任何 JBoss EAP 部署使用。

3.2.4. 从 ConfigSourceProvider 类获取 ConfigSource 配置

您可以创建并配置自定义 org.eclipse.microprofile.config.spi.ConfigSourceProvider 实现类，用于注册多个 ConfigSource 实例的实施。

流程

- 创建 **config-source-provider** ：

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

命令为名为 **org.example.MyConfigSourceProvider** 的实施类创建一个名为 **org.example.MyConfigSourceProvider** 的 **config-source-provider**，它由名为 **org.example** 的 **JBoss** 模块提供。

如果要使用 **org.example** 模块中的 **config-source-provider**，请将 **< module name="org.eclipse.microprofile.config.api"/>** 依赖项添加到 **path/to/org/example/main/module.xml** 文件。

此命令生成 **microprofile-config-smallrye** 子系统的以下 **XML** 配置：

```
<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>
```

ConfigSourceProvider 实施提供的属性可供任何 **JBoss EAP** 部署使用。

其他资源

- 有关如何向 **JBoss EAP** 服务器添加全局模块的详情，请参考 **JBoss EAP** *配置指南中的* 定义 全局模块。

3.3. MICROPROFILE FAULT TOLERANCE 配置

3.3.1. 添加 MicroProfile 容错扩展

MicroProfile Fault Tolerance 扩展包含在 **standalone-microprofile.xml** 和 **standalone-microprofile-ha.xml** 配置中，它们作为 **JBoss EAP XP** 的一部分提供。

该扩展不包括在标准的 **standalone.xml** 配置中。要使用扩展，您必须手动启用它。

先决条件

- 已安装 **EAP XP pack**。

流程

1. 使用以下管理 **CLI** 命令添加 **MicroProfile** 容错扩展：

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

2. 使用以下命令启用 **microprofile-fault-tolerance-smallrye** 子系统：

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

3. 使用以下管理命令重新载入服务器：

```
reload
```

3.4. MICROPROFILE HEALTH 配置

3.4.1. 使用管理 CLI 检查健康状况

您可以使用管理 **CLI** 检查系统健康状况。

流程

- 检查健康状况：

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

3.4.2. 使用管理控制台检查健康状况

您可以使用管理控制台检查系统健康状况。

检查运行时操作将健康检查和全局结果显示为布尔值。

流程

1. 导航到 **Runtime** 选项卡，再选择服务器。
2. 在 **Monitor** 列中，单击 **MicroProfile Health → View**。

3.4.3. 使用 HTTP 端点检查健康状况

健康检查会自动部署到 JBoss EAP 上的健康上下文中，因此您可以使用 HTTP 端点获取当前的健康状况。

可以从管理接口访问的 **/health** 端点的默认地址是 http://127.0.0.1:9990/health。

流程

- 要使用 HTTP 端点获取服务器的当前健康状况，请使用以下 URL：

```
http://<host>:<port>/health
```

访问此上下文以 JSON 格式显示健康检查，指示服务器是否健康。

3.4.4. 为 MicroProfile Health 启用身份验证

您可以将 健康 上下文配置为需要身份验证才能访问。

流程

1. 在 **microprofile-health-smallrye** 子系统中将 **security-enabled** 属性设置为 **true**。

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

2. 重新加载服务器以使更改生效。

```
reload
```

任何后续尝试访问 /health 端点都会触发身份验证提示符。

3.4.5. 决定服务器健康和就绪状态的就绪度探测

JBoss EAP XP 4.0.0 支持三个就绪度探测来确定服务器健康和就绪状态。

- server-status - 当 server-state 运行时 返回 UP。
- boot-errors - 当探测检测到没有引导错误时返回 UP。
- deployment-status - 当所有部署的状态为 OK 时返回 UP。

这些就绪度探测默认为启用。您可以使用 MicroProfile Config 属性 mp.health.disable-default-procedures 禁用探测。

以下示例演示了在 check 操作中使用三个探测：

```
[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"
        }
    ]
  }
}
```

其他资源

- JBoss EAP 中的 MicroProfile Health
- 没有定义探测时的全局状态

3.4.6. 没有定义探测时的全局状态

:empty-readiness-checks-status,:empty-liveness-checks-status, 和 :empty-startup-checks-status 管理属性指定没有 就绪度、存活度 或 启动探测 的全局状态。

这些属性允许应用报告 **'DOWN'，直到探测验证应用是否已就绪、实时或启动。默认情况下，应用程序会报告 'UP'。**

- **如果未定义就绪度探测，则 :empty- readiness -checks-status 属性指定 就绪度探测 的全局状态：**

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

- **如果未定义存活度探测，则 :empty- liveness -checks-status 属性指定 存活度探测 的全局状态：**

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

- **:empty-startup-checks-status 属性指定 启动探测 的全局状态，如果没有定义 启动探测 ：**

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

检查 **就绪度、存活度 和** 启动**探测 也考虑这些属性，/health HTTP 端点和 :check 操作也会考虑这些属性。**

您还可以修改这些属性，如下例所示：

```
/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 配置

3.5.1. 启用 microprofile-jwt-smallrye 子系统

MicroProfile JWT 集成由 microprofile-jwt-smallrye 子系统提供，包含在默认配置中。如果默认配置中没有子系统，您可以添加它，如下所示：

先决条件

- **EAP XP 已安装。**

流程

1. 在 JBoss EAP 中启用 MicroProfile JWT 小扩展：

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

2. 启用 microprofile-jwt-smallrye 子系统：

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

3. 重新载入服务器：

```
reload
```

microprofile-jwt-smallrye 子系统已启用。

3.6. MICROPROFILE 指标管理

3.6.1. 管理界面中可用的指标

JBoss EAP 子系统指标以 Prometheus 格式公开。

在 JBoss EAP 管理界面中自动提供指标，包括以下上下文：

- /metrics/ - 包含 MicroProfile 3.0 规范中指定的指标。
- /metrics/vendor - 包含特定于厂商的指标，如内存池。
- /metrics/application - 包含来自部署的应用和使用 MicroProfile Metrics API 的子系统的指标。

指标名称基于子系统和属性名称。例如，子系统 undertow 会公开应用部署中每个 servlet 的指标属性 request-count。此指标的名称是 jboss_undertow_request_count。前缀 jboss 标识 JBoss EAP 作为指标的来源。

3.6.2. 使用 HTTP 端点检查指标

使用 HTTP 端点，检查 JBoss EAP 管理界面上可用的指标。

流程

- 使用 curl 命令：

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

3.6.3. 为 MicroProfile 指标 HTTP 端点启用身份验证

配置 指标 上下文，要求用户被授权访问上下文。此配置扩展至 指标 上下文的所有子上下文。

流程

1. 在 microprofile-metrics-smallrye 子系统中将 security-enabled 属性设置为 true。

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

2. 重新加载服务器以使更改生效。

```
reload
```

任何后续尝试访问 metrics 端点都会生成身份验证提示符。

3.6.4. 获取 web 服务的请求数

获取公开其请求计数指标的 web 服务的请求数。

以下流程使用 helloworld-rs quickstart 作为 web 服务来获取请求计数。快速入门可从以下位置下载： jboss-eap-quickstarts。

先决条件

- Web 服务公开请求数。

流程

1. 为 **undertow** 子系统启用统计信息：

   - 启动启用统计的单机服务器：

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

   - 对于已在运行的服务器，启用 **undertow** 子系统的统计信息：

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

2. 部署 **helloworld-rs** 快速启动：

   - 在 **Quickstart** 的根目录中，使用 **Maven** 部署 **Web** 应用程序：

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

3. 使用 **curl** 命令在 **CLI** 中查询 **HTTP** 端点，并过滤 **request_count:**

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

   预期输出：

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

   返回的属性值是 **0.0**。

4. 在网页浏览器中访问位于 **http://localhost:8080/helloworld-rs/** 的快速入门并点击任何链接。

5. 再次通过 CLI 查询 HTTP 端点：

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

预期输出：

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

该值更新为 1.0。

重复最后两个步骤，以验证请求数是否已更新。

3.7. MICROPROFILE OPENAPI 管理

3.7.1. 启用 MicroProfile OpenAPI

microprofile-openapi-smallrye 子系统在 standalone-microprofile.xml 配置中提供。但是，JBoss EAP XP 默认使用 standalone.xml。您必须在 standalone.xml 中包含子系统才能使用它。

或者，您可以按照 使用 MicroProfile 子系统和扩展更新独立配置 的流程来更新 standalone.xml 配置文件。

流程

1. 在 JBoss EAP 中启用 MicroProfile OpenAPI smallrye 扩展：

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

2. 使用以下管理命令启用 microprofile-openapi-smallrye 子系统：

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

3. 重新加载服务器。

```
reload
```

microprofile-openapi-smallrye 子系统已启用。

3.7.2. 使用 Accept HTTP 标头请求 MicroProfile OpenAPI 文档

使用 Accept HTTP 标头从部署请求一个 MicroProfile OpenAPI 文档，采用 JSON 格式。

默认情况下，OpenAPI 端点会返回 YAML 文档。

先决条件

- **正在查询的部署被配置为返回 MicroProfile OpenAPI 文档。**

流程

- **发出以下 curl 命令查询部署的 /openapi 端点：**

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

使用部署的 URL 和端口替换 http://localhost:8080。

Accept 标头指示将使用 application/json 字符串返回 JSON 文档。

3.7.3. 使用 HTTP 参数请求 MicroProfile OpenAPI 文档

使用 HTTP 请求中的查询参数，从部署请求以 JSON 格式请求 MicroProfile OpenAPI 文档。

默认情况下，OpenAPI 端点会返回 YAML 文档。

先决条件

- **正在查询的部署被配置为返回 MicroProfile OpenAPI 文档。**

流程

- 发出以下 **curl** 命令查询部署的 **/openapi** 端点：

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

使用部署的 **URL** 和端口替换 **http://localhost:8080**。

HTTP 参数 **format=JSON** 表示要返回 **JSON** 文档。

3.7.4. 配置 JBoss EAP 以提供静态 OpenAPI 文档

配置 **JBoss EAP** 以提供描述主机的 **REST** 服务的静态 **OpenAPI** 文档。

当 **JBoss EAP** 配置为提供静态 **OpenAPI** 文档时，将在任何 **Jakarta RESTful Web** 服务和 **MicroProfile OpenAPI** 注解之前处理静态 **OpenAPI** 文档。

在生产环境中，在提供静态文档时禁用注解处理。禁用注解处理可确保客户端可以使用不可变和版本的 **API** 合同。

流程

1. 在应用程序源树中创建目录：

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

APPLICATION_ROOT 是包含应用的 **pom.xml** 配置文件的目录。

2. 查询 **OpenAPI** 端点，将输出重定向到文件：

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

默认情况下，端点提供 YAML 文档， format=JSON 指定返回 JSON 文档。

3. 配置应用程序，以便在处理 OpenAPI 文档模型时跳过注解扫描：

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

4. 重建应用程序：

```
$ mvn clean install
```

5. 使用以下管理 CLI 命令再次部署应用程序：

a. 取消部署应用程序：

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

b. 部署应用程序：

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

JBoss EAP 现在在 OpenAPI 端点上提供静态 OpenAPI 文档。

3.7.5. 禁用 microprofile-openapi-smallrye

您可以使用管理 CLI 在 JBoss EAP XP 中禁用 microprofile-openapi-smallrye 子系统。

流程

- 禁用 microprofile-openapi-smallrye 子系统：

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

3.8. MICROPROFILE REACTIVE 消息传递管理

3.8.1. 为 JBoss EAP 配置所需的 MicroProfile 被动消息传递扩展和子系统

如果要启用 JBoss EAP 实例的异步被动消息传递，您必须通过 JBoss EAP 管理 CLI 添加其扩展。

先决条件

- 您添加了带有 SmallRye 扩展和子系统的 Reactive Streams Operator。如需更多信息，请参阅 MicroProfile Reactive Streams Operators Subsystem Configuration: Required Extension。
- 您添加了使用 SmallRye 扩展和子系统的 Reactive Messaging。

流程

1. 打开 JBoss EAP 管理 CLI。
2. 输入以下代码：

```
[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"
    }
}
```

注意

如果您使用 Galleon 在 OpenShift 上置备服务器，请确保包含 microprofile-reactive-messaging Galleon 层来获取核心 MicroProfile 2.0.1 和被动消息传递功能，以及启用所需的子系统和扩展。请注意，此配置不包含启用 Kafka 连接器功能所需的 JBoss EAP 模块。为此，可使用 microprofile-reactive-messaging-kafka 层。

验证

如果您在管理 CLI 的两个位置中看到 成功，则已成功为 JBoss EAP 添加了所需的 MicroProfile 被动消息传递扩展和子系统。

提示

如果生成的代码显示 reload-required，您必须重新加载服务器配置以完全应用所有更改。要重新加载，请在单机服务器 CLI 中输入 reload。

3.9. 独立服务器配置

3.9.1. 独立服务器配置文件

JBoss EAP XP 包括额外的单机服务器配置文件 standalone-microprofile.xml 和 standalone-microprofile-ha.xml。

JBoss EAP 中包含的标准配置文件保持不变。请注意，JBoss EAP XP 4.0.0 不支持使用 domain.xml 文件或域模式。

表 3.1. JBoss EAP XP 中的独立配置文件

配置文件	用途	包括的功能	排除的功能
standalone.xml	这是启动单机服务器时使用的默认配置。	包括服务器的相关信息，包括子系统、网络、部署、套接字绑定和其他可配置的详细信息。	排除消息传递或高可用性所需的子系统。
standalone-microprofile.xml	此配置文件支持使用 MicroProfile 的应用。	包括服务器的相关信息，包括子系统、网络、部署、套接字绑定和其他可配置的详细信息。	排除以下功能： • Jakarta Enterprise Beans • 消息传递 • Jakarta EE Batch • jakarta Server Faces • Jakarta Enterprise Beans 计时器
standalone-ha.xml		包含默认子系统，并添加 **modcluster** 和 **jgroups** 子系统以实现高可用性。	排除消息传递所需的子系统。

配置文件	用途	包括的功能	排除的功能
standalone-microprofile-ha.xml	此独立文件支持使用 MicroProfile 的应用。	包括 **modcluster** 和 **jgroups** 子系统，以便在默认子系统之外实现高可用性。	排除消息传递所需的子系统。
standalone-full.xml		除了默认子系统外，还包括 **messaging-activemq** 和 **iiop-openjdk** 子系统。	
standalone-full-ha.xml	支持每个可能的子系统。	除了默认子系统外，还包括消息传递和高可用性的子系统。	
standalone-load-balancer.xml	支持使用内置 mod_cluster 前端负载平衡器对其他 JBoss EAP 实例进行负载平衡所需的最小子系统。		

默认情况下，将 JBoss EAP 作为单机服务器启动使用 standalone.xml 文件。若要使用独立 MicroProfile 配置启动 JBoss EAP，可使用 -c 参数。例如，

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

其它资源

- **启动和停止 JBoss EAP**
- **配置数据**

3.9.2. 使用 MicroProfile 子系统和扩展更新独立配置

您可以使用 docs/examples/enable-microprofile.cli 脚本使用 MicroProfile 子系统和扩展来更新标准单机服务器配置文件。enable-microprofile.cli 脚本旨在作为示例脚本，用于更新标准单机服务器配置文件，而不是自定义配置。

enable-microprofile.cli 脚本修改现有的单机服务器配置，并在独立配置文件中不存在以下 MicroProfile 子系统和扩展：

- **microprofile-config-smallrye**
- **microprofile-fault-tolerance-smallrye**
- **microprofile-health-smallrye**
- **microprofile-jwt-smallrye**
- **microprofile-metrics-smallrye**
- **microprofile-openapi-smallrye**
- **microprofile-opentracing-smallrye**

enable-microprofile.cli 脚本输出修改的高级描述。配置使用 elytron 子系统进行保护。security 子系统（如果存在）已从配置中删除。

先决条件

- **已安装 JBoss EAP XP。**

流程

1. **运行以下 CLI 脚本以更新默认的 standalone.xml 服务器配置文件：**

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

2. **使用以下命令，选择默认 standalone.xml 服务器配置文件以外的独立服务器配置：**

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

3.

指定的配置文件现在包含 MicroProfile 子系统和扩展。

第 4 章 为 JBOSS EAP 开发 MICROPROFILE 应用程序

4.1. MAVEN 和 JBOSS EAP MICROPROFILE MAVEN 存储库

4.1.1. 下载 JBoss EAP MicroProfile Maven 存储库补丁作为存档文件

每当为 JBoss EAP 发布 MicroProfile 扩展包时，都会为 JBoss EAP MicroProfile Maven 存储库提供相应的补丁。此补丁作为增量存档文件提供，该文件提取到现有的 Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven 存储库中。增量存档文件不会覆盖或删除任何现有文件，因此不需要回滚。

先决条件

- 您已 在红帽客户门户网站中设置了帐户。

流程

1. 打开浏览器并登录 红帽客户门户。
2. 从页面顶部的菜单中选择 **Downloads**。
3. 在列表中找到 **Red Hat JBoss Enterprise Application Platform** 条目并选择它。
4. 从 **Product** 下拉列表中，选择 **JBoss EAP XP**。
5. 从 **Version** 下拉列表中，选择 **4.0.0**。
6. 点 **Releases** 选项卡。
7. 在列表中找到 **JBoss EAP XP 4.0.0 Incremental Maven Repository**，然后单击 **Download**。
8. 将存档文件保存到您的本地目录。

其它资源

- 要了解有关 JBoss EAP Maven 存储库的更多信息，请参阅 JBoss EAP *开发指南中的* 关于 Maven 存储库。

4.1.2. 在本地系统上应用 JBoss EAP MicroProfile Maven 存储库补丁

您可以在本地文件系统上安装 JBoss EAP MicroProfile Maven 存储库补丁。

当您以增量存档文件的形式将补丁应用到存储库时，新文件将添加到此存储库中。增量存档文件不会覆盖或删除仓库上的任何现有文件，因此不需要回滚。

先决条件

- 您已在本地系统中 下载并安装 Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven 存储库。
  - 检查您是否在本地系统中安装了这个 Red Hat JBoss Enterprise Application Platform 7.4 Maven 存储库。
- 您已在本地系统中下载了 JBoss EAP XP 4.0.0 Incremental Maven 存储库。

流程

1. 找到 Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven 存储库的路径。例如： /path/to/repo/jboss-eap-7.4.0.GA-maven-repository/maven-repository/.
2. 将下载的 JBoss EAP XP 4.0.0 Incremental Maven 存储库直接提取到 Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven 存储库的目录中。例如，打开一个终端并发出以下命令，替换 Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven 存储库路径的值：

```
$ unzip -o jboss-eap-xp-4.0.0-incremental-maven-repository.zip -d
EAP_MAVEN_REPOSITORY_PATH
```

注意

EAP_MAVEN_REPOSITORY_PATH 指向 jboss-eap-7.4.0.GA-maven-repository。例如，这个过程演示了使用路径 /path/to/repo/jboss-eap-7.4.0.GA-maven-repository/。

将 JBoss EAP XP Incremental Maven 存储库提取到 Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven 存储库后，存储库名称将变为 JBoss EAP MicroProfile Maven 存储库。

其它资源

- 若要确定 JBoss EAP Maven 存储库的 URL，请参阅 JBoss EAP *开发指南中的* 确定 JBoss EAP Maven 存储库的 URL。

4.1.3. 支持的 JBoss EAP MicroProfile BOM

JBoss EAP XP 4.0.0 包括 JBoss EAP MicroProfile BOM。此 BOM 名为 jboss-eap-xp-microprofile，其用例支持 JBoss EAP MicroProfile API。

表 4.1. JBoss EAP MicroProfile BOM

BOM Artifact ID	使用案例
jboss-eap-xp-microprofile	此 BOM 的 **groupId** 是 **org.jboss.bom**，打包许多 JBoss EAP MicroProfile 支持的 API 依赖项，如 **microprofile-openapi-api** 和 **microprofile-config-api**。如果您使用这个 BOM，则不需要为受支持的 API 依赖项指定版本，因为 **jboss-eap-xp-microprofile** BOM 为依赖项指定这个值。

4.1.4. 使用 JBoss EAP MicroProfile Maven 存储库

在安装 Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven 存储库后，您可以访问 jboss-eap-xp-microprofile BOM，并将 JBoss EAP XP Incremental Maven 存储库应用到其中。存储库名称然后变为 JBoss EAP MicroProfile Maven 存储库。BOM 在 JBoss EAP XP Incremental Maven 存储库中提供。

您必须将以下之一配置为使用 JBoss EAP MicroProfile Maven 存储库：

- Maven 全局或用户设置

- 项目的 POM 文件

与共享服务器上的存储库管理器或存储库一起使用的 Maven 设置提供更好的项目控制和易管理性。

您可以使用替代镜像将特定存储库的所有查找请求重定向到存储库管理器，而无需更改项目文件。

警告

通过修改 POM 文件来配置 JBoss EAP MicroProfile Maven 存储库，覆盖所配置项目的全局和用户 Maven 设置。

先决条件

- 您已在本地系统上安装了 Red Hat JBoss Enterprise Application Platform 7.4 Maven 存储库，您已将 JBoss EAP XP Incremental Maven 存储库应用到其中。

流程

1. 选择配置方法并配置 JBoss EAP MicroProfile Maven 存储库。

2. 在配置了 JBoss EAP MicroProfile Maven 存储库后，将 jboss-eap-xp-microprofile BOM 添加到项目 POM 文件中。以下示例演示了如何在 pom.xml 文件的 < dependencyManagement& gt; 部分中配置 BOM：

```
<dependencyManagement>
  <dependencies>
    ...
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-xp-microprofile</artifactId>
      <version>4.0.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>
```

注意

如果您没有为 pom.xml 文件中的 type 元素指定值，Maven 会为元素指定一个 jar 值。

其它资源

- 有关选择配置 JBoss EAP Maven 存储库的方法的更多信息，请参阅 JBoss EAP *开发指南* 中的 使用 Maven 存储库。
- 有关管理依赖项的更多信息，请参阅 依赖管理。

4.2. MICROPROFILE 配置开发

4.2.1. 为 MicroProfile 配置创建 Maven 项目

创建一个具有所需依赖项的 Maven 项目，以及用于创建 MicroProfile Config 应用的目录结构。

先决条件

- 已安装 Maven。

流程

1. 设置 Maven 项目。

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

这将为项目和 pom.xml 配置文件创建目录结构。

2. 要让 POM 文件自动管理 MicroProfile 配置构件的版本和 jboss-eap-xp-microprofile BOM 中的 MicroProfile REST 客户端构件，请将 BOM 导入到项目 POM 文件的 &

lt;dependencyManagement > 部分。

```
<dependencyManagement>
  <dependencies>
    <!-- importing the microprofile BOM adds MicroProfile specs -->
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-xp-microprofile</artifactId>
      <version>4.0.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

3. 将 MicroProfile Config 构件和 MicroProfile REST 客户端构件和其他依赖项（由 BOM 管理）添加到项目 POM 文件的 <dependency > 部分。以下示例演示了将 MicroProfile 配置和 MicroProfile REST 客户端依赖项添加到该文件中：

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

4.2.2. 在应用中使用 MicroProfile Config 属性

创建使用配置的 ConfigSource 的应用。

先决条件

- JBoss EAP 中启用了 MicroProfile 配置。
- 已安装最新的 POM。
- Maven 项目已配置为创建 MicroProfile Config 应用。

流程

1. 创建用于存储类文件的目录：

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

其中 *APPLICATION_ROOT* 是包含应用的 pom.xml 配置文件的目录。

2. 进入新目录：

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

创建此流程中描述的所有类文件。

3. 创建名为 HelloApplication.java 的类文件，其内容如下：

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

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

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

}
```

此类将应用定义为 Jakarta RESTful Web Services 应用。

4. 创建名为 HelloService.java 的类文件，其内容如下：

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

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

5. 创建名为 HelloWorld.java 的类文件，其内容如下：

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

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

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

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

   @Inject
   HelloService helloService;

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

(1) MicroProfile Config 属性注入到类中，其注释为 @ConfigProperty (name="name", defaultValue="jim")。如果没有配置 ConfigSource，则返回值 jim。

6. 在 src/main/webapp/WEB-INF/ 目录中创建一个名为 beans.xml 的空文件：

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

其中 ***APPLICATION_ROOT*** 是包含应用的 pom.xml 配置文件的目录。

7. 进入应用程序的根目录：

```
$ cd APPLICATION_ROOT
```

其中 ***APPLICATION_ROOT*** 是包含应用的 pom.xml 配置文件的目录。

8. 构建项目：

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

9. 测试输出：

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

以下是预期的输出：

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

4.3. MICROPROFILE FAULT TOLERANCE 应用程序开发

4.3.1. 添加 MicroProfile 容错扩展

MicroProfile Fault Tolerance 扩展包含在 standalone-microprofile.xml 和 standalone-microprofile-ha.xml 配置中，它们作为 JBoss EAP XP 的一部分提供。

该扩展不包括在标准的 standalone.xml 配置中。要使用扩展，您必须手动启用它。

先决条件

- 已安装 EAP XP pack。

流程

1. 使用以下管理 CLI 命令添加 MicroProfile 容错扩展：

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

2. 使用以下命令启用 microprofile-fault-tolerance-smallrye 子系统：

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

3. 使用以下管理命令重新载入服务器：

```
reload
```

4.3.2. 为 MicroProfile Fault Tolerance 配置 Maven 项目

创建一个具有所需依赖项的 Maven 项目，以及用于创建 MicroProfile Fault Tolerance 应用的目录结构。

先决条件

- 已安装 Maven。

流程

1. 设置 Maven 项目：

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

该命令创建项目的目录结构以及 pom.xml 配置文件。

2. 要让 POM 文件自动管理 jboss-eap-xp-microprofile BOM 中 MicroProfile Fault Tolerance 工件的版本，请将 BOM 导入到项目 POM 文件的 & lt;dependencyManagement > 部分。

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

将 ${version.microprofile.bom} 替换为安装的 BOM 版本。

3. 将由 BOM 管理的 MicroProfile Fault Tolerance 工件添加到项目 POM 文件的 & lt;dependency > 部分。以下示例演示了将 MicroProfile Fault Tolerance 依赖项添加到该文件中：

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

4.3.3. 创建容错应用程序

创建一个容错应用程序，为容错实施重试、超时和回退模式。

先决条件

- 已配置了 Maven 依赖项。

流程

1. 创建用于存储类文件的目录：

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

APPLICATION_ROOT 是包含应用的 pom.xml 配置文件的目录。

2. 进入新目录：

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

对于以下步骤，请在新目录中创建所有类文件。

3. 创建一个简单的实体，其代表 Coffee 示例为 Coffee.java，其内容如下：

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

public class Coffee {

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

    public Coffee() {
    }

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

4. 创建包含以下内容的类文件 CoffeeApplication.java ：

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

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

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

5. 创建包含以下内容的 Jakarta 上下文和依赖注入 Bean 作为 CoffeeRepositoryService.java :

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

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

@ApplicationScoped
public class CoffeeRepositoryService {

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

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

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

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

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

6. 创建包含以下内容的类文件 CoffeeResource.java ：

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

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

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

    @Inject
    private CoffeeRepositoryService coffeeRepository;

    private AtomicLong counter = new AtomicLong(0);

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


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

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

    public List<Coffee> fallbackRecommendations(int id) {
        //always return a default coffee
        return Collections.singletonList(coffeeRepository.getCoffeeById(1));
    }
}
```

1

定义重新应用到 4 的数量。

2

以毫秒为单位定义超时间隔。

3

定义在调用失败时调用的回退方法。

7. 进入应用程序的根目录：

```
$ cd APPLICATION_ROOT
```

8. 使用以下 Maven 命令构建应用程序：

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

访问位于 http://localhost:8080/microprofile-fault-tolerance/coffee 的应用。

其它资源

- 有关容错应用的详细示例，包括用于测试应用的容错性故障，请参阅 microprofile-fault-tolerance Quickstart。

4.4. MICROPROFILE 健康开发

4.4.1. 自定义健康检查示例

microprofile-health-smallrye 子系统提供的默认实施执行基本的健康检查。如需更多详细信息，可能会包含在服务器或应用程序状态上。任何 Jakarta 上下文和依赖注入 Bean，其中包括 org.eclipse.microprofile.health.Liveness、org.eclipse.microprofile.health.Readiness、或 org.eclipse.microprofile.health.Startup 注释，它们在运行时会自动发现并调用。

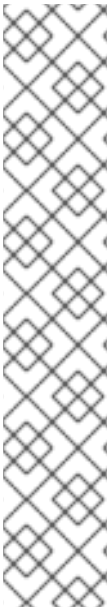
以下示例演示了如何创建返回 UP 状态的新健康检查实现。

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

import javax.enterprise.context.ApplicationScoped;
```

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

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

部署健康检查后，后续健康检查查询都包含自定义检查，如下例所示。

```
[standalone@localhost:9990 /] /subsystem=microprofile-health-smallrye:check
{
    "outcome" => "success",
    "result" => {
        "status" => "UP",
        "checks" => [
            {
                "name" => "deployments-status",
                "status" => "UP",
                "data" => {"<deployment_name>.war" => "OK"}
            },
            {
                "name" => "server-state",
                "status" => "UP",
                "data" => {"value" => "running"}
            },
            {
                "name" => "boot-errors",
                "status" => "UP"
            },
            {
                "name" => "health-test",
                "status" => "UP"
            },
            {
                "name" => "ready-deployment.<deployment_name>.war,
                "status" => "UP"
            },
            {
                "name" => "started-deployment.<deployment_name>.war",
                "status" => "UP"
            }
        ]
    }
}
```



注意

您可以使用以下命令进行存活度、就绪度和启动检查：

- /subsystem=microprofile-health-smallrye:check-live
- /subsystem=microprofile-health-smallrye:check-ready
- /subsystem=microprofile-health-smallrye:check-started

4.4.2. @Liveness 注释示例

以下示例演示了如何在应用中使用 @Liveness 注释。

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

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

4.4.3. @Readiness 注释示例

以下示例演示了如何检查与数据库的连接。如果数据库停机，就绪度检查会报告错误。

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

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

    @Override
    public HealthCheckResponse call() {
```

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

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

        return responseBuilder.build();
    }

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

4.4.4. @Startup 注释示例

以下是在应用中使用 @Startup 注释的示例：

```
@Startup
@ApplicationScoped
public class StartupHealthCheck implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.up("Application started");
    }
}
```

4.5. MICROPROFILE JWT 应用程序开发

4.5.1. 启用 microprofile-jwt-smallrye 子系统

MicroProfile JWT 集成由 microprofile-jwt-smallrye 子系统提供，包含在默认配置中。如果默认配置中没有子系统，您可以添加它，如下所示：

先决条件

- EAP XP 已安装。

流程

1. **在 JBoss EAP 中启用 MicroProfile JWT 小扩展：**

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

2. **启用 microprofile-jwt-smallrye 子系统：**

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

3. **重新载入服务器：**

```
reload
```

microprofile-jwt-smallrye 子系统已启用。

4.5.2. 配置 Maven 项目以开发 JWT 应用程序

创建一个具有所需依赖项的 Maven 项目，以及用于开发 JWT 应用的目录结构。

先决条件

- **已安装 Maven。**
- **启用 MicroProfile-jwt-smallrye 子系统。**

流程

1. **设置 maven 项目：**

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

该命令创建项目的目录结构以及 pom.xml 配置文件。

2. 要让 POM 文件自动管理 jboss-eap-xp-microprofile BOM 中 MicroProfile JWT 构件的版本，请将 BOM 导入到项目 POM 文件的 & lt;dependencyManagement > 部分。

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

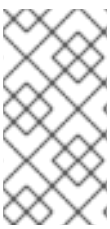
将 ${version.microprofile.bom} 替换为安装的 BOM 版本。

3. 将由 BOM 管理的 MicroProfile JWT 构件添加到项目 POM 文件的 & lt;dependency > 部分。以下示例演示了将 MicroProfile JWT 依赖项添加到文件中：

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

4.5.3. 使用 MicroProfile JWT 创建应用

创建一个应用，以基于 JWT 令牌验证请求，并根据令牌 bearer 的身份实施授权。



注意

以下流程提供了生成令牌的示例代码。对于生产环境，请使用红帽单点登录(SSO)等身份提供程序。

先决条件

- Maven 项目配置有正确的依赖项。

流程

1. 创建令牌生成器。

   此步骤充当参考。对于生产环境，请使用 Red Hat SSO 等身份提供程序。

   a. 为 token the generator 实用程序创建一个目录 src/test/java，并导航到它：

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

   b. 使用以下内容创建一个类文件 TokenUtil.java ：

```
package  com.example.mpjwt;

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

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

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

public class TokenUtil {

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

```
                .replaceAll("-----END (.*)----", "")
                .replaceAll("\r\n", "").replaceAll("\n", "").trim();

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

        return keyFactory.generatePrivate(keySpec);
    }
  }

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

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

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

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

    jwsObject.sign(signer);

    return jwsObject.serialize();
  }

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

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

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

2. 在 src/main/webapp/WEB-INF 目录中创建 web.xml 文件，其内容如下：

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

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

3. 使用以下内容创建一个类文件 SampleEndPoint.java ：

```
package com.example.mpjwt;

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

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

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

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

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

import org.eclipse.microprofile.jwt.JsonWebToken;

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

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

        return "Hello " + caller;
    }

    @Inject
```

```
JsonWebToken jwt;

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

    return helloReply;
}

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

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

        Period wait = today.until(next);

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

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

}

}
```

使用 @Path 注解的方法是 Jakarta RESTful Web Services 端点。

注释 @Claim 定义 JWT 声明。

4. 创建类文件 App.java 以启用 Jakarta RESTful Web Services：

```
package com.example.mpjwt;

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

import org.eclipse.microprofile.auth.LoginConfig;

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

注释 @LoginConfig (authMethod="MP-JWT", realmName="MP JWT Realm") 在部署过程中启用 JWT RBAC。

5. 使用以下 Maven 命令编译应用程序：

```
$ mvn package
```

6. 使用令牌生成器工具生成 JWT 令牌：

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

7. 使用以下 Maven 命令构建和部署应用程序：

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

8. 测试应用。

- 使用 bearer 令牌调用 Sample/subscription 端点：

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

- 调用 Sample/birthday 端点：

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

4.6. MICROPROFILE 指标开发

4.6.1. 创建 MicroProfile 指标应用

创建应用，以返回向应用发出的请求数量。

流程

1. 使用以下内容创建一个类文件 HelloService.java ：

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

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

2. 使用以下内容创建一个类文件 HelloWorld.java ：

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

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

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

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

3. 更新 pom.xml 文件，使其包含以下依赖项：

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

4. **使用以下 Maven 命令构建应用程序：**

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

5. **测试指标：**

   a. **在 CLI 中运行以下命令：**

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

   预期输出：

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

   b. **在浏览器中，导航到 URL http://localhost:8080/helloworld-rs/rest/json。**

   c. **在 CLI 中重新尝试以下命令：**

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

   预期输出：

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

4.7. 开发 MICROPROFILE OPENAPI 应用

4.7.1. 启用 MicroProfile OpenAPI

microprofile-openapi-smallrye 子系统在 standalone-microprofile.xml 配置中提供。但是，JBoss EAP XP 默认使用 standalone.xml。您必须在 standalone.xml 中包含子系统才能使用它。

或者，您可以按照 使用 MicroProfile 子系统和扩展更新独立配置 的流程来更新 standalone.xml 配置文件。

流程

1. 在 JBoss EAP 中启用 MicroProfile OpenAPI smallrye 扩展：

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

2. 使用以下管理命令启用 microprofile-openapi-smallrye 子系统：

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

3. 重新加载服务器。

```
reload
```

microprofile-openapi-smallrye 子系统已启用。

4.7.2. 为 MicroProfile OpenAPI 配置 Maven 项目

创建一个 Maven 项目，以设置用于创建 MicroProfile OpenAPI 应用的依赖项。

先决条件

- 已安装 Maven。
- JBoss EAP Maven 存储库已配置。

流程

1. **初始化项目：**

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

该命令创建项目的目录结构以及 pom.xml 配置文件。

2. **编辑 pom.xml 配置文件使其包含：**

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

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>com.example.microprofile.openapi</groupId>
   <artifactId>microprofile-openapi</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>war</packaging>

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

   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <maven.compiler.source>1.8</maven.compiler.source>
      <maven.compiler.target>1.8</maven.compiler.target>
      <version.server.bom>4.0.0.GA</version.server.bom>
   </properties>

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

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

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

使用 pom.xml 配置文件和目录结构来创建应用。

其他资源

- 有关配置 JBoss EAP Maven 存储库的详情，请参考使用 POM 文件 配置 JBoss EAP Maven 存储库。

4.7.3. 创建 MicroProfile OpenAPI 应用

创建返回 OpenAPI v3 文档的应用程序。

先决条件

- 配置了 Maven 项目以创建 MicroProfile OpenAPI 应用。

流程

1. 创建用于存储类文件的目录：

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

APPLICATION_ROOT 是包含应用的 pom.xml 配置文件的目录。

2. 进入新目录：

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

必须在这个目录中创建以下步骤中的所有类文件。

3. 使用以下内容创建类文件 InventoryApplication.java ：

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

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

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

此类充当应用的 REST 端点。

4. 使用以下内容创建一个类文件 Fruit.java ：

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

public class Fruit {

    private final String name;
    private final String description;

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

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

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

5. 创建包含以下内容的类文件 FruitResource.java ：

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

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

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

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

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

    public FruitResource() {
```

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

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

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

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

6. 进入应用程序的根目录：

```
$ cd APPLICATION_ROOT
```

7. 使用以下 Maven 命令构建和部署应用程序：

```
$ mvn wildfly:deploy
```

8. 测试应用。

- 使用 curl 访问示例应用程序的 OpenAPI 文档：

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

- 返回以下输出：

```
openapi: 3.0.1
info:
  title: Archetype Created Web Application
  version: "1.0"
servers:
- url: /microprofile-openapi
```

```
paths:
  /inventory/fruit:
    get:
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Fruit'
    post:
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Fruit'
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Fruit'
    delete:
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Fruit'
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Fruit'
components:
  schemas:
    Fruit:
      type: object
      properties:
        description:
          type: string
        name:
          type: string
```

其它资源

- 有关 MicroProfile SmallRye OpenAPI 中定义的注解列表，请参阅 MicroProfile OpenAPI

注解。

4.7.4. 配置 JBoss EAP 以提供静态 OpenAPI 文档

配置 JBoss EAP 以提供描述主机的 REST 服务的静态 OpenAPI 文档。

当 JBoss EAP 配置为提供静态 OpenAPI 文档时，将在任何 Jakarta RESTful Web 服务和 MicroProfile OpenAPI 注解之前处理静态 OpenAPI 文档。

在生产环境中，在提供静态文档时禁用注解处理。禁用注解处理可确保客户端可以使用不可变和版本的 API 合同。

流程

1. 在应用程序源树中创建目录：

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

APPLICATION_ROOT 是包含应用的 pom.xml 配置文件的目录。

2. 查询 OpenAPI 端点，将输出重定向到文件：

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

默认情况下，端点提供 YAML 文档， format=JSON 指定返回 JSON 文档。

3. 配置应用程序，以便在处理 OpenAPI 文档模型时跳过注解扫描：

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

4. 重建应用程序：

```
$ mvn clean install
```

5. 使用以下管理 CLI 命令再次部署应用程序：

   a. 取消部署应用程序：

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

   b. 部署应用程序：

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

JBoss EAP 现在在 OpenAPI 端点上提供静态 OpenAPI 文档。

4.8. MICROPROFILE REST 客户端开发

4.8.1. MicroProfile REST 客户端和 Jakarta RESTful Web 服务语法的比较

MicroProfile REST 客户端启用分布式对象通信版本，它也在 CORBA、Java Remote Method Invocation (RMI)、JBoss Remoting Project 和 RESTEasy 中实施。例如，考虑资源：

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

以下示例演示了使用 Jakarta RESTful Web Services-native 方法访问 TestResource 类：

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

但是，Microprofile REST 客户端通过直接调用 test （） 方法来支持更直观的语法，如下例所示：

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

```
    public String test();
}

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

在前面的示例中，对 TestResource 类进行调用会更容易地使用 TestResourceIntf 类，如调用 service.test （） 所示。

以下示例是 TestResourceIntf 类的更详细版本：

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

调用 service.test ("p", "q", "e") 方法会导致 HTTP 信息，如下例所示：

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

e
```

4.8.2. 在 MicroProfile REST 客户端中编程注册提供程序

通过 MicroProfile REST 客户端，您可以通过注册提供程序来配置客户端环境。例如：

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

4.8.3. MicroProfile REST 客户端中的提供程序声明注册

通过向目标接口添加 org.eclipse.microprofile.rest.client.annotation.RegisterProvider 注释，以声明性方式注册提供程序，如下例所示：

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

使用注释声明 MyClientResponseFilter 类和 MyMessageBodyReader 类无需调用 RestClientBuilder.register （） 方法。

4.8.4. MicroProfile REST 客户端中的标头声明规格

您可以使用以下方法为 HTTP 请求指定标头：

- 通过注解一个资源方法参数。
- 按照声明使用 org.eclipse.microprofile.rest.client.annotation.ClientHeaderParam 注释。

以下示例演示了使用注解 @HeaderParam 标注其中一个资源方法参数来设置标头：

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

以下示例演示了如何使用 org.eclipse.microprofile.rest.client.annotation.ClientHeaderParam 注释设置标头：

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

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

4.8.5. MicroProfile REST 客户端中的 ResponseExceptionMapper

org.eclipse.microprofile.rest.client.ext.ResponseExceptionMapper 类是 javax.ws.rs.ext.ExceptionMapper 类的 client-sideverse，它在 Jakarta RESTful Web Services 中定义。ExceptionMapper.toResponse （） 方法在服务器端处理 响应 类期间将引发 Exception 类。Response Exception Mapper.toThrowable （） 方法将在客户端上收到的 Response 类将 HTTP 错误状态转换为例外类。

您可以以编程方式或声明性方式注册 ResponseExceptionMapper 类。如果没有注册的 ResponseExceptionMapper 类，默认的 ResponseExceptionMapper 类会将状态为 >= 400 的任何响应映射到 WebApplicationException 类。

4.8.6. 使用 MicroProfile REST 客户端进行上下文依赖项注入

使用 MicroProfile REST 客户端时，您必须使用 @RegisterRestClient 类为作为 Jakarta 上下文和依赖项注入(Jakarta 上下文和依赖注入) bean 标注任何接口。例如：

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

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

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

在这里，MicroProfile REST 客户端实施为 TestDataBase 类服务创建一个客户端，允许由 TestResourceImpl 类轻松访问。但是，它不包括 TestDataBase 类实施的路径的信息。此信息可以通过

可选的 @RegisterProvider 参数 baseUri 提供：

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

这表示您可以访问位于 https://localhost:8080/webapp 的 TestDataBase 的实现。您还可以使用 MicroProfile 配置向外部提供信息：

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

例如，以下属性表示您可以访问位于 https://localhost:8080/webapp 的 com.bluemonkeydiamond.TestDatabase 类的实现：

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

您可以为 Jakarta 上下文和依赖注入客户端提供许多其他属性。例如，com.mycompany.remoteServices.MyServiceClient/mp-rest/providers，以逗号分隔的、要包含在客户端中的完全限定提供程序类名称的列表。

其他资源

- 有关 MicroProfile REST 客户端规范的更多信息，请参阅关于 MicroProfile 的 Rest 客户端。
- 有关 MicroProfile REST 客户端 2.0 功能的更多信息，请参阅 MicroProfile REST 客户端 2.0。

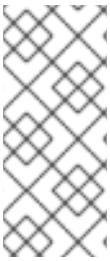
第 5 章 在 JBOSS EAP XP 的 OPENSHIFT 镜像上构建并运行微服务应用程序

您可以在 OpenShift 镜像的 JBoss EAP XP 上构建并运行微服务应用程序。

注意

JBoss EAP XP 仅支持 OpenShift 4 及更新的版本。

使用以下工作流，使用 source-to-image (S2I)进程为 JBoss EAP XP 构建并运行微服务应用。



注意

JBoss EAP XP 4.0.0 的 OpenShift 镜像提供默认的单机配置文件，该文件基于 standalone-microprofile-ha.xml 文件。有关 JBoss EAP XP 中包含的服务器配置文件的更多信息，请参阅 *单机服务器配置文件* 部分。

此工作流使用 microprofile-config quickstart 作为示例。Quickstart 提供了一个小型特定工作示例，可用作您自己的项目的参考。如需更多信息，请参阅 JBoss EAP XP 4.0.0 附带的 microprofile-config Quickstart。

其他资源

- 有关 JBoss EAP XP 中包含的服务器配置文件的更多信息，请参阅 单机服务器配置文件。

5.1. 为应用程序部署准备 OPENSHIFT

为应用部署准备 OpenShift。

先决条件

已安装可正常运行的 OpenShift 实例。如需更多信息，请参阅红帽客户门户网站中的 *安装和配置 OpenShift Container Platform 集群* 一书。

流程

1. 使用 oc login 命令登录到您的 OpenShift 实例。

2. 在 OpenShift 中创建一个新项目。

   项目允许一组用户独立于其他组组织和管理内容。您可以使用以下命令在 OpenShift 中创建项目。

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

   例如，对于 microprofile-config Quickstart，使用以下命令创建一个名为 eap-demo 的新项目：

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

5.2. 为 RED HAT CONTAINER REGISTRY 配置身份验证

在导入并使用 JBoss EAP XP 的 OpenShift 镜像之前，您必须配置身份验证到 Red Hat Container Registry。

使用 registry 服务帐户创建身份验证令牌，以配置对 Red Hat Container Registry 的访问。使用身份验证令牌时，您不需要在 OpenShift 配置中使用或存储您的红帽帐户的用户名和密码。

流程

1. 按照红帽客户门户网站中的说明，使用 Registry Service Account 管理应用程序 创建身份验证令牌。

2. 下载包含令牌的 OpenShift 机密的 YAML 文件。

   您可以从令牌的 Token Information 页面中的 OpenShift Secret 选项卡下载 YAML 文件。

3. 使用您下载的 YAML 文件为 OpenShift 项目创建身份验证令牌 secret：

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

4.

使用以下命令，为 OpenShift 项目配置 secret，将示例中的机密名称替换为上一步中创建的机密的名称。

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

其他资源

- 为 Red Hat Container Registry 配置身份验证
- registry Service Account 管理应用程序
- 配置对安全 registry 的访问

5.3. 为 JBOSS EAP XP 导入最新的 OPENSHIFT 镜像流和模板

为 JBoss EAP XP 导入最新的 OpenShift 镜像流和模板。

重要

OpenShift 上的 OpenJDK 8 镜像和镜像流已弃用。

OpenShift 中仍然支持镜像和镜像流。但是，不会对这些镜像和镜像流进行任何增强，以后可能会删除它们。红帽继续根据标准支持条款和条件提供全面支持和程序错误修复 OpenJDK 8 镜像和镜像流。

流程

1. 要将 JBoss EAP XP 的 OpenShift 镜像的最新镜像流和模板导入到 OpenShift 项目的命名空间，请使用以下命令：

   a. 导入 JDK 11 镜像流：

```
oc replace --force -f https://raw.githubusercontent.com/jboss-container-images/jboss-eap-openshift-templates/eap-xp4/eap-xp4-openjdk11-image-stream.json
```

此命令导入以下镜像流和模板：

- JDK 11 builder 镜像流：jboss-eap-xp4-openjdk11-openshift
- JDK 11 运行时镜像流：jboss-eap-xp4-openjdk11-runtime-openshift

b. 导入 OpenShift 模板：

```
oc replace --force -f https://raw.githubusercontent.com/jboss-container-images/jboss-eap-openshift-templates/eap-xp4/templates/eap-xp4-basic-s2i.json
```

注意

使用上述命令导入的 JBoss EAP XP 镜像流和模板仅在该 OpenShift 项目中可用。

2. 如果您有对常规 openshift 命名空间的管理权限，并希望所有项目可以访问镜像流和模板，请将 -n openshift 添加到 命令的 oc replace 行中。例如：

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

3. 如果要将镜像流和模板导入到不同的项目中，请将 -n *PROJECT_NAME* 添加到 命令的 oc replace 行中。例如：

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

如果使用 cluster-samples-operator，请参阅有关配置集群样本 Operator 的 OpenShift 文档。有关配置集群示例 Operator 的详细信息，请参阅配置 Cluster Samples Operator。

5.4. 在 OPENSHIFT 上部署 JBOSS EAP XP SOURCE-TO-IMAGE (S2I)应用程序

在 OpenShift 上部署 JBoss EAP XP Source-to-image (S2I)应用。

先决条件

- 可选：模板可以为许多模板参数指定默认值，您可能需要覆盖一些或全部默认值。要查看模板信息，包括参数列表和任何默认值，请使用 **oc describe template TEMPLATE_NAME** 命令。

流程

1. 使用 JBoss EAP XP 镜像和 Java 应用程序的源代码创建一个新的 OpenShift 应用。将提供的 JBoss EAP XP 模板之一用于 S2I 构建。

```
$ oc new-app --template=eap-xp4-basic-s2i \ 1
 -p EAP_IMAGE_NAME=jboss-eap-xp4-openjdk11-openshift:latest \
 -p EAP_RUNTIME_IMAGE_NAME=jboss-eap-xp4-openjdk11-runtime-openshift:latest \
 -p IMAGE_STREAM_NAMESPACE=eap-demo \ 2
 -p SOURCE_REPOSITORY_URL=https://github.com/jboss-developer/jboss-eap-quickstarts
\ 3
 -p SOURCE_REPOSITORY_REF=xp-4.0.x \ 4
 -p CONTEXT_DIR=microprofile-config 5
```

1. 要使用的模板。应用程序镜像使用 **latest** 标签标记。
2. 最新镜像流和模板 导入到项目的命名空间 中，因此您必须指定查找镜像流的命名空间。这通常是项目的名称。
3. 包含应用源代码的存储库的 URL。
4. 用于源代码的 Git 存储库引用。这可以是 Git 分支或标签引用。
5. 要构建的源存储库中的目录。

注意

模板可以为许多模板参数指定默认值，您可能需要覆盖一些或全部默认值。要查看模板信息，包括参数列表和任何默认值，请使用 **oc describe template *TEMPLATE_NAME*** 命令。

在创建新的 OpenShift 应用时，您可能还想 配置环境变量。

2. 检索构建配置的名称。

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

3. 使用上一步中的构建配置的名称来查看构建的 Maven 进度。

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

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

例如，对于 **microprofile-config**，以下命令显示了 Maven 构建的进度。

```
$ oc logs -f buildconfig/eap-xp4-basic-app-build-artifacts

…
Push successful
$ oc logs -f buildconfig/eap-xp4-basic-app
…
Push successful
```

其他资源

- 为 JBoss EAP XP 导入最新的 OpenShift 镜像流和模板。
- 为应用部署准备 OpenShift。

5.5. 完成 JBOSS EAP XP SOURCE-TO-IMAGE (S2I)应用程序的部署后任务

根据您的应用，您可能需要在构建和部署 OpenShift 应用后完成一些任务。

部署后任务示例包括：

- 公开服务，以便应用可从 OpenShift 外部查看。
- 将应用程序扩展到特定数量的副本。

流程

1. 使用以下命令获取应用程序的服务名称。

```
$ oc get service
```

2. 可选 ：将主服务作为路由公开，以便您可以从 OpenShift 外部访问应用。例如，对于 microprofile-config Quickstart，请使用以下命令公开所需的服务和端口。

注意

如果您使用模板来创建应用程序，则路由可能已存在。如果存在，请继续下一步。

```
$ oc expose service/eap-xp4-basic-app --port=8080
```

3. 获取路由的 URL。

```
$ oc get route
```

4. 使用 URL 在 Web 浏览器中访问应用程序。URL 是上一命令输出中的 HOST/PORT 字段的值。

注意

对于 JBoss EAP XP 4.0.0 GA 发布，Microprofile 配置快速启动不会回复对应用程序的根上下文的 HTTPS GET 请求。此功能增强仅适用于 {JBossXPShortName101} GA 发行版。

例如，若要与 Microprofile Config 应用交互，您的浏览器中 URL 可能是 http://*HOST_PORT_Value*/config/value。

如果您的应用程序没有使用 JBoss EAP root 上下文，请将应用的上下文附加到 URL。例如，对于 microprofile-config Quickstart，URL 可以是 http://*HOST_PORT_VALUE*/microprofile-config/。

5. 另外，您可以通过运行以下命令来扩展应用程序实例。此命令将副本数增加到 3。

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

例如，对于 microprofile-config Quickstart，请使用以下命令来扩展应用：

```
$ oc scale deploymentconfig/eap-xp4-basic-app --replicas=3
```

其它资源

有关 JBoss EAP XP Quickstarts 的更多信息，请参阅 JBoss EAP XP 快速入门。

第 6 章 功能修剪

在构建可引导 JAR 时，您可以决定要包含哪些 JBoss EAP 功能和子系统：

注意

只有 OpenShift 或构建可引导 JAR 时，才支持修剪功能。

其他资源

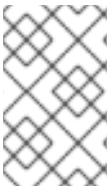
- 关于可引导 JAR

6.1. 可用的 JBOSS EAP 层

红帽提供了多个层，可在 OpenShift 或可引导 JAR 中自定义 JBoss EAP 服务器的调配。

三个层是提供核心功能的基本层。其他层是 decorator 层，其增强了基本层的功能。

大多数 decorator 层可用于在 JBoss EAP 中为 OpenShift 构建 S2I 镜像，或构建可引导 JAR。几个层不支持 S2I 镜像；层的描述中指出这个限制。



注意

仅支持列出的层。不支持此处列出的层。

6.1.1. 基本层

每个基础层包括典型的服务器用户用例的核心功能。

datasources-web-server

此层包含一个 servlet 容器，以及配置数据源的功能。

此层不包括 MicroProfile 功能。

此层支持以下 Jakarta EE 规格：

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

jaxrs-server

此层通过以下 JBoss EAP 子系统增强了 datasources-web-server 层：

- jaxrs
- weld
- jpa

此层还在容器中本地添加了基于 Infinispan 的第二级实体缓存。

以下 MicroProfile 功能包含在此层中：

- MicroProfile REST 客户端

除了 datasources-web-server 层支持的那些层，还需要以下 Jakarta EE 规格：

- jakarta 上下文和依赖注入 2.0
- Jakarta Bean Validation 2.0
- Jakarta Interceptors 1.2
- jakarta RESTful Web Services 2.1
- Jakarta Persistence 2.2

cloud-server

此层通过以下 JBoss EAP 子系统增强了 jaxrs-server 层：

- resource-adapters

- messaging-activemq （远程代理消息传递，非嵌入式消息传递）

此层还在 jaxrs-server 层中添加以下可观察性功能：

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

除了 jaxrs-server 层支持的外，该层还支持以下 Jakarta EE 规格：

- jakarta Security 1.0

6.1.2. decorator 层

Decorator 层不会独立使用。您可以使用基本层配置一个或多个 decorator 层，以提供额外的功能。

ejb-lite

此 decorator 层向置备的服务器添加最小的 Jakarta Enterprise Beans 实施。这个层不包括以下支持：

- IIOP 集成
- MDB 实例池
- 远程连接器资源

只有在构建可引导 JAR 时，才支持此层。使用 S2I 时不支持这个层。

Jakarta Enterprise Beans

此 decorator 层扩展 ejb-lite 层。除了 ejb-lite 层中包含的基本功能外，此层还对置备的服务器添加以下支持：

- MDB 实例池
- 远程连接器资源

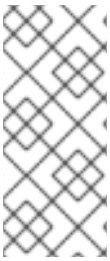
如果要使用消息驱动的 Bean (MDB)或 Jakarta Enterprise Beans 远程功能，则使用这个层。如果不需要这些功能，请使用 ejb-lite 层。

只有在构建可引导 JAR 时，才支持此层。使用 S2I 时不支持这个层。

ejb-local-cache

此 decorator 层将 Jakarta Enterprise Beans 的本地缓存支持添加到置备的服务器。

依赖项：只有当您包含 ejb-lite 层或 ejb 层时，才能包含此层。



注意

此层与 ejb-dist-cache 层不兼容。如果包含 ejb-dist-cache 层，则无法包含 ejb-local-cache 层。如果您同时包含两个层，则生成的构建可能会包含意外的 Jakarta Enterprise Beans 配置。

只有在构建可引导 JAR 时，才支持此层。使用 S2I 时不支持这个层。

ejb-dist-cache

此 decorator 层将 Jakarta Enterprise Beans 的分布式缓存支持添加到置备的服务器。

依赖项：只有当您包含 ejb-lite 层或 ejb 层时，才能包含此层。

注意

此层与 ejb-local-cache 层不兼容。如果包含 ejb-dist-cache 层，则无法包含 ejb-local-cache 层。如果您同时包含两个层，则生成的构建可能会导致意外配置。

只有在构建可引导 JAR 时，才支持此层。使用 S2I 时不支持这个层。

jdr

此 decorator 层添加 JBoss 诊断报告(jdr)子系统，以便在从红帽请求支持时收集诊断数据。

只有在构建可引导 JAR 时，才支持此层。使用 S2I 时不支持这个层。

jakarta Persistence

此 decorator 层为单节点服务器添加持久性功能。请注意，只有在服务器能够组成集群时，分布式缓存才可以正常工作。

该层将 Hibernate 库添加到置备的服务器，并具有以下支持：

- jpa 子系统的配置
- infinispan 子系统的配置
- 本地 Hibernate 缓存容器

注意

此层与 jpa-distributed 层不兼容。如果包含 jpa 层，则无法包含 jpa-distributed 层。

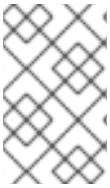
只有在构建可引导 JAR 时，才支持此层。使用 S2I 时不支持这个层。

jpa-distributed

这个 decorator 层为集群中运行的服务器添加持久性功能。该层将 Hibernate 库添加到置备的服务

器，并具有以下支持：

- jpa 子系统的配置
- infinispan 子系统的配置
- 本地 Hibernate 缓存容器
- Invalidation 和 replication Hibernate 缓存容器
- 配置 jgroups 子系统



注意

此层与 jpa 层不兼容。如果包含 jpa 层，则无法包含 jpa-distributed 层。

只有在构建可引导 JAR 时，才支持此层。使用 S2I 时不支持这个层。

jakarta Server Faces

此 decorator 层将 jsf 子系统添加到调配的服务器。

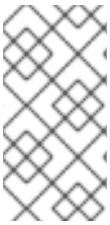
只有在构建可引导 JAR 时，才支持此层。使用 S2I 时不支持这个层。

microprofile-platform

此 decorator 层向置备的服务器添加以下 MicroProfile 功能：

- MicroProfile Config
- MicroProfile Fault Tolerance

- MicroProfile Health
- MicroProfile JWT
- MicroProfile Metrics
- MicroProfile OpenAPI
- MicroProfile OpenTracing



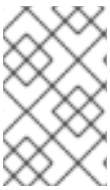
注意

此层包括 可观察 层中包含的 MicroProfile 功能。如果包含这个层，则不需要包含 可观察 层。

Observability（可观察性）

这个 decorator 层在置备的服务器中添加了以下可观察性功能：

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



注意

此层内置于 cloud-server 层。您不需要将此层添加到 cloud-server 层。

remote-activemq

此 decorator 层添加了与远程 ActiveMQ 代理通信到调配的服务器的功能，集成消息传递支持。

池的连接工厂配置将 guest 指定为 用户和密码 属性的值。您可以使用 CLI 脚本在运行时更改这些值。

只有在构建可引导 JAR 时，才支持此层。使用 S2I 时不支持这个层。

sso

这种 decorator 层将 Red Hat Single Sign-On 集成到置备的服务器。

只有在使用 S2I 调配服务器时，才应使用此层。

web-console

这个 decorator 层将管理控制台添加到置备的服务器。

只有在构建可引导 JAR 时，才支持此层。使用 S2I 时不支持这个层。

web-clustering

此 decorator 层通过为适合集群环境的数据会话处理配置基于本地 Infinispan 的容器 Web 缓存来添加对可分布式 Web 应用的支持。

web-passivation

此 decorator 层通过为适合单一节点环境的数据会话配置本地基于 Infinispan 的容器 Web 缓存来添加对可分布式 Web 应用程序的支持。

只有在构建可引导 JAR 时，才支持此层。使用 S2I 时不支持这个层。

webservices

此层为调配的服务器添加 Web 服务功能，支持 Jakarta Web 服务部署。

只有在构建可引导 JAR 时，才支持此层。使用 S2I 时不支持这个层。

其他资源

- **池连接工厂属性**

第 7 章 在红帽 CODEREADY STUDIO 上为 JBOSS EAP 启用 MICROPROFILE 应用程序开发

如果要在 CodeReady Studio 上开发的应用程序中纳入 MicroProfile 功能，您必须在 CodeReady Studio 中启用对 JBoss EAP 的 MicroProfile 支持。

JBoss EAP 扩展包提供对 MicroProfile 的支持。

JBoss EAP 7.2 及更早版本不支持 JBoss EAP expansion pack。

JBoss EAP 扩展包的每个版本都支持 JBoss EAP 的特定补丁。详情请查看 JBoss EAP 扩展软件包支持和生命周期政策页。

重要

用于 Openshift 的 JBoss EAP XP Quickstarts 仅作为技术预览提供。技术预览功能不包括在红帽生产服务级别协议（SLA）中，且其功能可能并不完善。因此，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

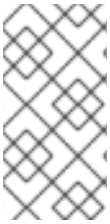
如需有关 技术预览功能支持范围 的信息，请参阅红帽客户门户网站中的技术预览功能支持范围。

7.1. 配置 CODEREADY STUDIO 以使用 MICROPROFILE 功能

要在 JBoss EAP 上启用 MicroProfile 支持，请为 JBoss EAP XP 注册一个新的运行时服务器，然后创建新的 JBoss EAP 7.4 服务器。

为服务器指定适当的名称，可帮助您识别它支持 MicroProfile 功能。

此服务器使用新创建的 JBoss EAP XP 运行时，该运行时指向之前安装的运行时，并使用 standalone-microprofile.xml 配置文件。



注意

如果您在 Red Hat CodeReady Studio 中将 Target runtime 设为 7.4 或更新的运行时版本，则您的项目与 Jakarta EE 8 规范兼容。

先决条件

- 已安装 JBoss EAP XP 4.0.0。

流程

1. 在 新建服务器 对话框中设置新的服务器。
   a. 在 Select server type 列表中，选择 *Red Hat JBoss Enterprise Application Platform 7.4*。
   b. 在 Server 的主机名 字段中，输入 *localhost*。
   c. 在 Server name 字段中，输入 *JBoss EAP 7.4 XP*。
   d. 点击 Next。
2. 配置新的服务器。
   a. 在 Home directory 字段中，如果您不想使用默认设置，请指定一个新目录；例如：*home/myname/dev/microprofile/runtimes/jboss-eap-7.4*。
   b. 确保 执行环境 设置为 *JavaSE-1.8*。
   c. 可选：更改 Server 基础目录 和配置文件字段中的值。
   d. 点 Finish。

结果

现在，您已准备好开始使用 MicroProfile 功能开发应用程序，或开始使用 JBoss EAP 的 MicroProfile 快速入门。

7.2. 为 CODEREADY STUDIO 使用 MICROPROFILE 快速入门

启用 MicroProfile 快速入门可简化在已安装的服务器上运行和测试的简单示例。

这些示例演示了以下 MicroProfile 功能。

- MicroProfile Config
- MicroProfile Fault Tolerance
- MicroProfile Health
- MicroProfile JWT
- MicroProfile Metrics
- MicroProfile OpenAPI
- MicroProfile OpenTracing
- MicroProfile REST 客户端

流程

1. 从 Quickstart Parent Artifact 导入 pom.xml 文件。

2. 如果您使用的快速入门需要环境变量，请配置环境变量。

   在服务器 Overview 对话框中在启动配置中定义环境变量。

   例如，microprofile-opentracing quickstart 使用以下环境变量：

   - JAEGER_REPORTER_LOG_SPANS 设置为 true
   - JAEGER_SAMPLER_PARAM 设置为 1
   - JAEGER_SAMPLER_TYPE 设置为 const

其他资源

关于 Microprofile

关于 JBoss Enterprise Application Platform expansion pack

Red Hat JBoss Enterprise Application Platform expansion pack 支持和生命周期政策

第 8 章 可引导 JAR

您可以使用 JBoss EAP JAR Maven 插件构建和打包微服务应用，作为可引导 JAR。然后，您可以在 JBoss EAP 裸机平台或 JBoss EAP OpenShift 平台上运行应用程序。

8.1. 关于可引导 JAR

您可以使用 JBoss EAP JAR Maven 插件构建和打包微服务应用，作为可引导 JAR。

可引导 JAR 包含服务器、打包的应用程序以及启动服务器所需的运行时。

JBoss EAP JAR Maven 插件使用 Galleon 修剪功能来减少服务器的大小和内存占用量。因此，您可以根据要求配置服务器，仅包含提供您需要的功能的 Galleon 层。

JBoss EAP JAR Maven 插件支持执行 JBoss EAP CLI 脚本文件来自定义服务器配置。CLI 脚本包含用于配置服务器的 CLI 命令列表。

可引导 JAR 类似于标准 JBoss EAP 服务器：

- 它支持 JBoss EAP 通用管理 CLI 命令。
- 它可以通过 JBoss EAP 管理控制台进行管理。

在可引导 JAR 中打包服务器时存在以下限制：

- 不支持需要服务器重启的 CLI 管理操作。
- 服务器不能以 admin-only 模式重启，这是启动与服务器管理相关的服务的模式。
- 如果您关闭服务器，则应用到服务器的更新将会丢失。

另外，您还可以置备 hollow bootable JAR。此 JAR 仅包含服务器，以便您可以重复使用服务器来运行不同的应用。

其他资源

有关功能修剪的详情，请参考 Capability Trimming。

8.2. JBOSS EAP MAVEN 插件

您可以使用 JBoss EAP JAR Maven 插件构建应用作为可引导 JAR。

您可以从 Maven 存储库检索最新的 Maven 插件版本，该版本位于 /ga/org/wildfly/plugins/wildfly-jar-maven-plugin 的索引。

在 Maven 项目中，src 目录包含构建应用程序所需的所有源文件。在 JBoss EAP JAR Maven 插件构建可引导 JAR 后，生成的 JAR 位于 target/<application>-bootable.jar 中。

JBoss EAP JAR Maven 插件还提供以下功能：

- 将 CLI 脚本命令应用到服务器。
- 使用 org.jboss.eap:wildfly-galleon-pack Galleon 功能 pack 及其某些层来自定义服务器配置文件。
- 支持将额外文件添加到打包的可引导 JAR 中，如密钥存储文件。
- 包含创建 hollow bootable JAR 的功能，即不包含应用的可引导 JAR。

在使用 JBoss EAP JAR Maven 插件创建可引导 JAR 后，您可以通过发出以下命令来启动应用：将 target/myapp-bootable.jar 替换为可引导 JAR 的路径。例如：

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

注意

要获取支持的可引导 JAR 启动命令列表，请在启动命令末尾附加 --help。例如，java -jar target/myapp-bootable.jar --help。

其他资源

- 有关支持的 JBoss EAP Galleon 层的详情，请参考 可用的 JBoss EAP 层。
- 有关为项目构建功能软件包支持的 Galleon 插件的详情，请参考 WildFly Galleon Maven 插件文档。
- 有关选择配置 JBoss EAP Maven 存储库的方法的信息，请参阅使用 Maven 存储库。
- 有关 Maven 项目目录的详情，请参考 *Apache Maven* 文档中的 标准目录布局简介。

8.3. 可引导 JAR 参数

查看下表中的参数，以了解用于可引导 JAR 的支持参数。

表 8.1. 支持的可引导 JAR 可执行文件参数

参数	描述
--help	显示指定命令的帮助信息并退出。
--cli-script=<path>	指定启动可引导 JAR 时执行的 JBoss CLI 脚本的路径。如果指定的路径相对，则路径会根据用于启动可引导 JAR 的 Java 虚拟机实例的工作目录解析。
--deployment=<path>	特定于 hollow bootable JAR 的参数。指定包含您要部署到服务器上的应用程序的 WAR、JAR、EAR 文件或展开目录的路径。
--display-galleon-config	输出生成的 Galleon 配置文件的内容。

参数	描述
--install-dir=<path>	默认情况下，JVM 设置用于在启动可引导 JAR 后创建 *TEMP* 目录。您可以使用 **--install-dir** 参数指定安装服务器的目录。
-secmgr	运行安装有安全管理器的服务器。
-b<interface>=<value>	将系统属性 **jboss.bind.address.<interface>** 设置为给定值。例如，**bmanagement=IP_ADDRESS**。
-b=<value>	设置系统属性 **jboss.bind.address**，用于为公共接口配置绑定地址。如果没有指定值，则默认为 127.0.0.1。
-D<name>[=<value>]	指定服务器运行时设置的系统属性。可引导 JAR JVM 不会设置这些系统属性。
--properties=<url>	从指定的 URL 加载系统属性。
-S<name>[=value]	设置安全属性。
-u=<value>	设置系统属性 **jboss.default.multicast.address**，用于在配置文件中的 socket-binding 元素中配置多播地址。如果没有指定值，则默认为 230.0.0.4。
--version	显示应用服务器版本并退出。

8.4. 为可引导 JAR 服务器指定 GALLEON 层

您可以指定 Galleon 层来为服务器构建自定义配置。另外，您可以指定您要从服务器中排除的 Galleon 层。

要引用单个功能软件包，请使用 < feature-pack-location& gt; 元素来指定其位置。以下示例在 Maven 插件配置文件的 < feature-pack-location> 项中指定 org.jboss.eap:wildfly-galleon-pack:4.0.0.GA-

redhat-00002。

```
<configuration>
  <feature-pack-location>org.jboss.eap:wildfly-galleon-pack:4.0.0.GA-redhat-00002</feature-
pack-location>
</configuration>
```

如果您需要引用多个功能包，请在 < feature-packs> 元素中 列出它们。以下示例显示了将 Red Hat Single Sign-On 功能 pack 添加到 < feature-packs> 元素中：

```
<configuration>
    <feature-packs>
        <feature-pack>
            <location>org.jboss.eap:wildfly-galleon-pack:4.0.0.GA-redhat-00002</location>
        </feature-pack>
        <feature-pack>
            <location>org.keycloak:keycloak-adapter-galleon-pack:15.0.4.redhat-00001</location>
        </feature-pack>
    </feature-packs>
</configuration>
```

您可以组合来自多个功能软件包的 Galleon 层，将可引导 JAR 服务器配置为只包含提供您需要的功能的 Galleon 层。

注意

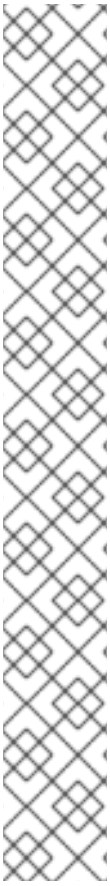
在裸机平台上，如果您在配置文件中没有指定 Galleon 层，则调配的服务器包含一个与默认 standalone-microprofile.xml 配置相同的配置。

在 OpenShift 平台上，在插件配置中添加了 & lt;cloud /> 配置元素后，您选择不要在配置文件中指定 Galleon 层，置备的服务器包含针对云环境调整的配置，类似于默认的 standalone-microprofile-ha.xml。

先决条件

- **已安装 Maven。**
- **您已检查了最新的 Maven 插件版本，如 *MAVEN_PLUGIN_VERSION*.X.GA.Final-redhat-00001，其中 *MAVEN_PLUGIN_VERSION* 是主版本，*X* 是 microversion。请参阅 /ga/org/wildfly/plugins/wildfly-jar-maven-plugin 的索引。**
-

您已检查了最新的 Galleon 功能软件包版本，如 4.0.X.GA-redhat-*BUILD_NUMBER*，其中 *X* 是 JBoss EAP XP 的微版本，*BUILD_NUMBER* 是 Galleon 功能软件包的构建号。*X* 和 *BUILD_NUMBER* 可以在 JBoss EAP XP 4.0.0 产品生命周期中演进。请参阅 /ga/org/jboss/eap/wildfly-galleon-pack 的索引。



注意

流程中显示的示例指定以下属性：

- ${bootable.jar.maven.plugin.version} 用于 Maven 插件版本。
- ${jboss.xp.galleon.feature.pack.version} 用于 Galleon 功能软件包版本。

您必须在项目中设置这些属性。例如：

```
<properties>
   <bootable.jar.maven.plugin.version>6.1.2.Final-redhat-
00001</bootable.jar.maven.plugin.version>
   <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-
00002</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

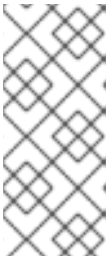
1. 确定支持的 JBoss EAP Galleon 层，提供运行应用程序所需的功能。

2. 在 Maven 项目 pom.xml 文件的 &lt ;plugin& gt; 元素中引用 JBoss EAP feature pack 位置。您必须指定任何 Maven 插件的最新版本，以及 org.jboss.eap:wildfly-galleon-pack Galleon 功能软件包的最新版本，如下例所示。以下示例还显示包含单个功能包，其中包括 jaxrs-server 基础层和 jpa 分布式 层。jaxrs-server 基础层为服务器提供额外的支持。

```
<plugins>
    <plugin>
        <groupId>org.wildfly.plugins</groupId>
         <artifactId>wildfly-jar-maven-plugin</artifactId>
         <version>${bootable.jar.maven.plugin.version}</version>
        <configuration>
            <feature-pack-location>org.jboss.eap:wildfly-galleon-
pack:${jboss.xp.galleon.feature.pack.version}</feature-pack-location>
            <layers>
               <layer>jaxrs-server</layer>
               <layer>jpa-distributed</layer>
```

```
            </layers>
            <excluded-layers>
               <layer>jpa</layer>
            </excluded-layers>
            ...
</plugins>
```

本例还显示来自项目的 jpa 层的排除。



注意

如果您在项目中包含 jpa-distributed 层，则必须从 jaxrs-server 层中排除 jpa 层。jpa 层配置本地 infinispan hibernate 缓存，而 jpa-distributed 层配置远程 infinispan hibernate 缓存。

其他资源

- 有关可用基本层的详情，请参考 基本层。
- 有关为项目构建功能软件包支持的 Galleon 插件的详情，请参考 WildFly Galleon Maven 插件文档。
- 有关选择配置 JBoss EAP Maven 存储库的方法的信息，请参阅 Maven 和 JBoss EAP MicroProfile Maven 存储库。
- 有关管理您的 Maven 依赖项的详情，请参考 *Apache Maven 项目* 文档中的 依赖 管理。

8.5. 在 JBOSS EAP 裸机平台上使用可引导 JAR

您可以将应用打包为 JBoss EAP 裸机平台上的可引导 JAR。

注意

- 要在 JBoss EAP 裸机平台上构建可引导 JAR 时使用自定义 Galleon 功能包和层，请参阅为 JBoss EAP 构建和使用自定义 Galleon 层。
- 使用 oc new-build 命令构建应用程序镜像时，请确保使用此 S2I 构建器镜像 jboss-eap-xp4-openjdk11-openshift:latest，而不是 jboss-eap74-openjdk11-openshift:latest。

可引导 JAR 包含服务器、打包的应用程序以及启动服务器所需的运行时。

此流程演示了使用 JBoss EAP JAR Maven 插件将 MicroProfile Config 微服务应用打包为可引导 JAR。请参阅 MicroProfile 配置开发。

您可以使用 CLI 脚本在可引导 JAR 打包期间配置服务器。

重要

在构建必须打包在可引导 JAR 中的 web 应用程序时，您必须在 pom.xml 文件的 <packaging > 项中指定 war。例如：

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

此值需要将构建应用打包为 WAR 文件，而不是作为默认的 JAR 文件。

在仅构建 hollow bootable JAR 的 Maven 项目中，将打包值设置为 pom。例如：

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

当您为 Maven 项目构建 hollow bootable JAR 时，您不限于使用 pom 打包。您可以通过在 < hollow-jar > 项中为任何类型的打包指定 true 来创建 true，如 war。请参阅在 JBoss EAP 裸机平台上创建休眠 JAR。

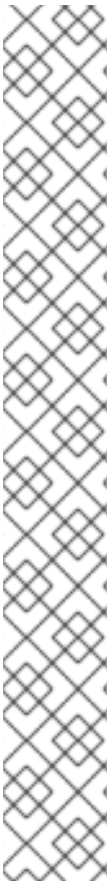
先决条件

- 您已检查了最新的 Maven 插件版本，如 *MAVEN_PLUGIN_VERSION*.X.GA.Final-redhat-

00001，其中 *MAVEN_PLUGIN_VERSION* 是主版本，*X* 是 microversion。请参阅 /ga/org/wildfly/plugins/wildfly-jar-maven-plugin 的索引。

- 您已检查了最新的 Galleon 功能软件包版本，如 4.0.X.GA-redhat-*BUILD_NUMBER*，其中 *X* 是 JBoss EAP XP 的微版本，*BUILD_NUMBER* 是 Galleon 功能软件包的构建号。*X* 和 *BUILD_NUMBER* 可以在 JBoss EAP XP 4.0.0 产品生命周期中演进。请参阅 /ga/org/jboss/eap/wildfly-galleon-pack 的索引。

- 您已创建了 Maven 项目，设置父依赖项，并添加了用于创建 MicroProfile 应用的依赖项。请参阅 MicroProfile 配置开发。



注意

流程中显示的示例指定以下属性：

- ${bootable.jar.maven.plugin.version} 用于 Maven 插件版本。
- ${jboss.xp.galleon.feature.pack.version} 用于 Galleon 功能软件包版本。

您必须在项目中设置这些属性。例如：

```
<properties>
   <bootable.jar.maven.plugin.version>6.1.2.Final-redhat-
00001</bootable.jar.maven.plugin.version>
   <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-
00002</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

1. 在 pom.xml 文件的 <build > 元素中添加以下内容。您必须指定任何 Maven 插件的最新版本，以及 org.jboss.eap:wildfly-galleon-pack Galleon 功能软件包的最新版本。例如：

```
<plugins>
   <plugin>
      <groupId>org.wildfly.plugins</groupId>
      <artifactId>wildfly-jar-maven-plugin</artifactId>
      <version>${bootable.jar.maven.plugin.version}</version>
      <configuration>
         <feature-pack-location>org.jboss.eap:wildfly-galleon-
```

```
pack:${jboss.xp.galleon.feature.pack.version}</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>
```

注意

如果您没有在 pom.xml 文件中指定 Galleon 层，则可引导 JAR 服务器包含一个与 standalone-microprofile.xml 配置相同的配置。

2. **将应用程序打包为可引导 JAR:**

```
$ mvn package
```

3. **启动应用程序：**

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

注意

该示例使用 NAME 作为环境变量，但您可以选择使用 jim，这是默认值。

注意

要查看支持的可引导 JAR 参数的列表，请将 --help 附加到 java -jar target/microprofile-config-bootable.jar 命令的末尾。

4. **在 Web 浏览器中指定以下 URL 以访问 MicroProfile Config 应用：**

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

5. *验证 ：* 在终端中运行以下命令来测试应用程序的行为：

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

以下是预期的输出：

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

其他资源

- 有关可用 MicroProfile 配置功能的信息，请参阅 MicroProfile 配置。
- 有关 ConfigSources 的信息，请参阅 MicroProfile 配置参考。

8.6. 在 JBOSS EAP 裸机平台上创建休眠 JAR

您可以在 JBoss EAP 裸机平台上将应用程序打包为 hollow bootable JAR。

hollow bootable JAR 仅包含 JBoss EAP 服务器。hollow bootable JAR 由 JBoss EAP JAR Maven 插件打包。该应用在服务器运行时提供。如果您需要为不同的应用重新使用服务器配置，则 hollow bootable JAR 非常有用。

先决条件

- 您已创建了 Maven 项目，设置父依赖项，并添加了用于创建应用程序的依赖项。请参阅 MicroProfile 配置开发。
- 您已在 JBoss EAP 裸机平台上使用可引导 JAR 完成了 pom.xml 文件配置步骤。
- 您已检查了最新的 Maven 插件版本，如 *MAVEN_PLUGIN_VERSION*.X.GA.Final-redhat-00001，其中 *MAVEN_PLUGIN_VERSION* 是主版本，*X* 是 microversion。请参阅 /ga/org/wildfly/plugins/wildfly-jar-maven-plugin 的索引。
- 您已检查了最新的 Galleon 功能软件包版本，如 4.0.X.GA-redhat-*BUILD_NUMBER*，其中

X 是 JBoss EAP XP 的微版本，*BUILD_NUMBER* 是 Galleon 功能软件包的构建号。*X* 和 *BUILD_NUMBER* 可以在 JBoss EAP XP 4.0.0 产品生命周期中演进。请参阅 /ga/org/jboss/eap/wildfly-galleon-pack 的索引。

注意

此流程中显示的示例为 Galleon 功能软件包版本指定 ${jboss.xp.galleon.feature.pack.version}，但您必须在项目中设置属性。例如：

```
<properties>
   <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-
00002</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

1. 要构建 hollow bootable JAR，您必须在项目 pom.xml 文件中将 < hollow-jar > 插件配置元素设置为 true。例如：

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

注意

通过在 < hollow-jar& gt; 元素中指定 true，JBoss EAP JAR Maven 插件不会将应用包含在 JAR 中。

1. 构建 hollow bootable JAR:

```
$ mvn clean package
```

2. 运行 hollow bootable JAR:

```
$ java -jar target/microprofile-config-bootable.jar --deployment=target/microprofile-config.war
```

重要

要指定要在服务器上部署的 WAR 文件的路径，请使用以下参数，其中 < PATH_NAME& gt; 是部署的路径。

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

3. 访问应用程序：

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

注意

要将 Web 应用注册到根目录，请将应用命名为 ROOT.war。

其他资源

- **有关可用 MicroProfile 功能的信息，请参阅 MicroProfile 配置。**
- **有关 JBoss EAP XP 4.0.0 支持的 JBoss EAP JAR Maven 插件的更多信息，请参阅 JBoss EAP Maven 插件。**

8.7. 构建时执行的 CLI 脚本

您可以创建 CLI 脚本，以在可引导 JAR 打包期间配置服务器。

CLI 脚本是一个文本文件，其中包含可用于应用其他服务器配置的 CLI 命令序列。例如，您可以创建一个脚本来添加新的日志记录器到 logging 子系统。

您还可以在 CLI 脚本中指定更复杂的操作。例如，您可以将安全管理操作分组到单个命令中，以便为管理 HTTP 端点启用 HTTP 身份验证。

注意

在将应用程序打包为可引导 JAR 之前，您必须在插件 配置的 <cli-session > 元素中定义 CLI 脚本。这样可确保服务器配置设置在打包可引导 JAR 后保留。

虽然您可以组合预定义的 Galleon 层来配置部署应用程序的服务器，但存在限制。例如，在打包可引导 JAR 时，您无法使用 Galleon 层启用 HTTPS undertow 侦听器。反之，您必须使用 CLI 脚本。

您必须在 pom.xml 文件的 &lt ;cli-session& gt; 元素中定义 CLI 脚本。下表显示了 CLI 会话属性的类型：

表 8.2. CLI 脚本属性

参数	描述
script-files	脚本文件的路径列表。
properties-file	可选属性，用于指定属性文件的路径。此文件列出了脚本可以使用 **${my.prop}** 语法来引用的 Java 属性。以下示例将 **public inet-address** 设置为 **all.addresses** 的 值：**/interface=public:write-attribute (name=inet-address,value=${all.addresses})**
resolve-expressions	包含布尔值的可选属性。指明在向服务器发送操作请求前是否解析系统属性或表达式。默认值为 **true**。

注意

- CLI 脚本会按照 pom.xml 文件的 &lt ;cli-session > 元素定义的顺序启动。
- JBoss EAP JAR Maven 插件为每个 CLI 会话启动嵌入式服务器。因此，您的 CLI 脚本不必启动或停止嵌入式服务器。

8.8. 在运行时执行 CLI 脚本

您可以在运行时对服务器配置应用更改；这为您提供了针对执行上下文调整服务器的灵活性。但是，对服务器应用更改的首选方法是在构建期间应用。

流程

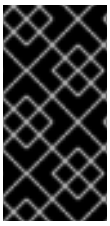
- 启动可引导 JAR 和 --cli-script 参数。

  例如：

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

注意

- CLI 脚本必须是文本文件(UTF-8)，如果存在，则文件扩展是无意义的，但建议 .cli 扩展。
- 需要服务器重启的操作将终止您的可引导 JAR 实例。
- 连接、重新加载、关闭 以及与嵌入式服务器 和补丁 相关的任何命令都不可操作。
- 不支持在 admin-mode 中无法执行的 CLI 命令，如 jdbc-driver-info。

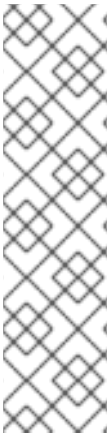


重要

如果您在不执行 CLI 脚本的情况下重新启动服务器，您的新服务器实例将不包含来自上一服务器实例的更改。

8.9. 在 JBOSS EAP OPENSHIFT 平台上使用可引导 JAR

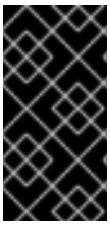
将应用打包为可引导 JAR 后，您可以在 JBoss EAP OpenShift 平台上运行应用。



注意

- 要在 JBoss EAP OpenShift 平台上构建可引导 JAR 时使用自定义 Galleon 功能包和层，请参阅为 JBoss EAP 构建和使用自定义 Galleon 层。

- 使用 oc new-build 命令构建应用程序镜像时，请确保使用此 S2I 构建器镜像 jboss-eap-xp4-openjdk11-openshift:latest，而不是 jboss-eap74-openjdk11-openshift:latest。



重要

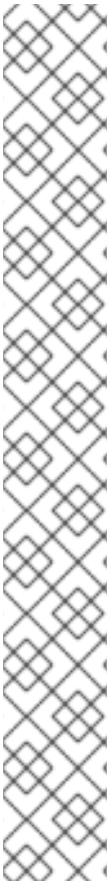
在 OpenShift 上，您无法将 EAP Operator 自动事务恢复功能与可引导 JAR 搭配使用。计划在以后的 JBoss EAP XP 4.0.0 补丁版本中修复此技术限制。

先决条件

- 您已为 MicroProfile 配置开发 创建了 Maven 项目。

- 您已检查了最新的 Maven 插件版本，如 *MAVEN_PLUGIN_VERSION*.X.GA.Final-redhat-00001，其中 *MAVEN_PLUGIN_VERSION* 是主版本，*X* 是 microversion。请参阅 /ga/org/wildfly/plugins/wildfly-jar-maven-plugin 的索引。

- 您已检查了最新的 Galleon 功能软件包版本，如 4.0.X.GA-redhat-*BUILD_NUMBER*，其中 *X* 是 JBoss EAP XP 4 的微版本，*BUILD_NUMBER* 是 Galleon 功能软件包的构建号。*X* 和 *BUILD_NUMBER* 可以在 JBoss EAP XP 4.0.0 产品生命周期中演进。请参阅 /ga/org/jboss/eap/wildfly-galleon-pack 的索引。



注意

流程中显示的示例指定以下属性：

- ${bootable.jar.maven.plugin.version} 用于 Maven 插件版本。
- ${jboss.xp.galleon.feature.pack.version} 用于 Galleon 功能软件包版本。

您必须在项目中设置这些属性。例如：

```
<properties>
  <bootable.jar.maven.plugin.version>6.1.2.Final-redhat-
00001</bootable.jar.maven.plugin.version>
  <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-
00002</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

1. 在 pom.xml 文件的 <build > 元素中添加以下内容。您必须指定任何 Maven 插件的最新版本，以及 org.jboss.eap:wildfly-galleon-pack Galleon 功能软件包的最新版本。例如：

```
<plugins>
  <plugin>
    <groupId>org.wildfly.plugins</groupId>
    <artifactId>wildfly-jar-maven-plugin</artifactId>
    <version>${bootable.jar.maven.plugin.version}</version>
    <configuration>
      <feature-pack-location>org.jboss.eap:wildfly-galleon-
pack:${jboss.xp.galleon.feature.pack.version}</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>
```

注意

您必须在插件配置的 < configuration> 元素中包含 <cloud /> 元素，因此 JBoss EAP Maven JAR 插件可以识别您选择的 OpenShift 平台。

2. **打包应用程序：**

```
$ mvn package
```

3. **使用 oc login 命令登录到您的 OpenShift 实例。**

4. **在 OpenShift 中创建一个新项目。例如：**

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

5. **输入以下 oc 命令来创建应用程序镜像：**

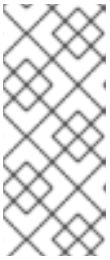
```
$ mkdir target/openshift && cp target/microprofile-config-bootable.jar target/openshift (1)

$ oc import-image ubi8/openjdk-11 --from=registry.redhat.io/ubi8/openjdk-11 --confirm (2)

$ oc new-build --strategy source --binary --image-stream openjdk-11 --name microprofile-config-app (3)

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

(1) **在目标目录中创建 openshift 子目录。打包的应用程序被复制到创建的子目录中。**

(2) **将最新的 OpenJDK 11 镜像流标签和镜像信息导入到 OpenShift 项目中。**

(3) **基于 microprofile-config-app 目录和 OpenJDK 11 镜像流创建构建配置。**

(4)



注意

OpenShift 将一组 CLI 脚本命令应用到可引导 JAR 配置文件，将其调整为云环境。您可以通过在 Maven 项目 / target 目录中 打开 bootable-jar-build-artifacts/generated-cli-script.txt 文件来访问此脚本。

6. *验证：*

运行以下命令，查看可用的 OpenShift pod 列表并检查 pod 构建状态：

```
$ oc get pods
```

验证构建的应用程序镜像：

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

输出显示了构建的应用镜像详细信息，如名称和镜像存储库、标签等。对于此流程中的示例，镜像流名称和标签输出显示 microprofile-config-app:latest。

7. **部署应用程序：**

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

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

重要

要为可引导 JAR 提供系统属性，您必须使用 JAVA_OPTS_APPEND 环境变量。以下示例演示了使用 JAVA_OPTS_APPEND 环境变量：

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

新建应用已创建并启动。应用程序配置作为新服务公开。

8. *验证* ：在终端中运行以下命令来测试应用程序的行为：

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

预期输出：

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

其他资源

- 有关 MicroProfile 的信息，请参阅 MicroProfile 配置。
- 有关 ConfigSources 的信息，请参阅 默认的 MicroProfile 配置属性。

8.10. 为 OPENSHIFT 配置可引导 JAR

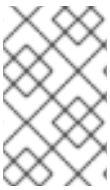
在使用可引导 JAR 之前，您可以配置 JVM 设置，以确保单机服务器在 JBoss EAP for OpenShift 上正确运行。

使用 JAVA_OPTS_APPEND 环境变量来配置 JVM 设置。使用 JAVA_ARGS 命令，为可引导 JAR 提供参数。

您可以使用环境变量为属性设置值。例如，您可以使用 JAVA_OPTS_APPEND 环境变量将 -Dwildfly.statistics-enabled 属性设置为 true ：

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

现在为您的服务器启用统计。



注意

如果您需要为可引导 JAR 提供参数，请使用 JAVA_ARGS 环境变量。

用于 OpenShift 的 JBoss EAP 提供 JDK 11 镜像。要运行与可引导 JAR 关联的应用，您必须首先将

最新的 OpenJDK 11 镜像流标签和镜像信息导入到 OpenShift 项目中。然后，您可以使用环境变量在导入的镜像中配置 JVM。

您可以应用相同的配置选项来配置用于 JBoss EAP for OpenShift S2I 镜像的 JVM，但有以下区别：

- 可选： -Xlog 功能不可用，但您可以通过启用 -Xlog:gc 来设置垃圾回收日志记录。例如： JAVA_OPTS_APPEND="-Xlog:gc*:file=/tmp/gc.log:time"。
- 要增加初始元空间大小，您可以设置 GC_METASPACE_SIZE 环境变量。为获得最佳元数据容量性能，请将值设为 96。
- 要更好地生成随机文件，请使用 JAVA_OPTS_APPEND 环境变量将 java.security.egd 属性设置为 -Djava.security.egd=file:/dev/urandom。

这些配置在导入的 OpenJDK 11 镜像上运行时，改进了 JVM 的内存设置和垃圾回收功能。

8.11. 在 OPENSHIFT 上使用应用程序中的 CONFIGMAP

对于 OpenShift，您可以使用部署控制器(dc)将 configmap 挂载到用于运行应用的 pod 中。

ConfigMap 是一个 OpenShift 资源，用于将非机密数据存储在键值对中。

在指定 microprofile-platform Galleon 层来添加 microprofile-config-smallrye 子系统以及服务器配置文件的任何扩展后，您可以使用 CLI 脚本向服务器配置添加新的 ConfigSource。您可以将 CLI 脚本保存在可访问的目录中，如 /scripts 目录，位于 Maven 项目的根目录中。

MicroProfile 配置功能在 JBoss EAP 中使用 SmallRye Config 组件实施，它由 microprofile-config-smallrye 子系统提供。此子系统包含在 microprofile-platform Galleon 层中。

先决条件

- 您已安装了 Maven。

- 您已配置了 JBoss EAP Maven 存储库。

- 您已将应用打包为可引导 JAR，您可以在 JBoss EAP OpenShift 平台上运行应用程序。有关在 OpenShift 平台上构建应用程序作为可引导 JAR 的详情，请参考在 JBoss EAP OpenShift 平台上使用可引导 JAR。

流程

1. 在项目的根目录下，创建名为 scripts 的目录。例如：

```
$ mkdir scripts
```

2. 创建 cli.properties 文件，并将文件保存到 /scripts 目录中。在此文件中定义 config.path 和 config.ordinal 系统属性。例如：

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

3. 创建一个 CLI 脚本，如 mp-config.cli，并将它保存在可引导 JAR 的可访问目录中，如 /scripts 目录。以下示例显示了 mp-config.cli 脚本的内容：

```
# config map

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

mp-config.cli CLI 脚本会创建一个新的 ConfigSource，该脚本将从属性文件检索到的 ordinal 和 path 值。

4. 将脚本保存到 /scripts 目录中，该目录位于项目的根目录。

5. 在现有插件 <configuration> 元素 中添加以下 配置提取：

```
<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. **打包应用程序：**

```
$ mvn package
```

7. **使用 oc login 命令登录到您的 OpenShift 实例。**

8. ***可选：*** **如果您之前还没有创建 target/openshift 子目录，则必须发出以下命令来创建 suddirectory：**

```
$ mkdir target/openshift
```

9. **将打包的应用复制到创建的子目录中。**

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

10. **使用 target/openshift 子目录作为二进制输入来构建应用程序：**

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

注意

OpenShift 将一组 CLI 脚本命令应用到可引导 JAR 配置文件，为云环境启用它。您可以通过在 Maven 项目 /target 目录中打开 bootable-jar-build-artifacts/generated-cli-script.txt 文件来访问此脚本。

11. **创建 ConfigMap。例如：**

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

12. **使用 dc 将 ConfigMap 挂载到应用。例如：**

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

执行 oc set volume 命令后，应用程序将使用新的配置设置重新部署。

13. 测试输出：

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

以下是预期的输出：

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

其他资源

- 有关 MicroProfile ConfigSources 属性的信息，请参阅 默认的 MicroProfile 配置属性。
- 有关可引导 JAR 参数的详情，请参考 支持的可引导 JAR 参数。

8.12. 创建可引导 JAR MAVEN 项目

按照以下步骤创建示例 Maven 项目。您必须先创建一个 Maven 项目，然后才能执行以下步骤：

- 为您的可引导 JAR 启用 JSON 日志记录
- 为多个可引导 JAR 实例启用 Web 会话数据存储
- 使用 CLI 脚本为可引导 JAR 启用 HTTP 身份验证
- 使用红帽单点登录保护您的 JBoss EAP 可引导 JAR 应用

在项目 pom.xml 文件中，您可以配置 Maven 来检索构建可引导 JAR 所需的项目工件。

流程

1. 设置 Maven 项目：

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

其中 *GROUP_ID* 是项目的 groupId，*ARTIFACT_ID* 是项目的 artifactId。

2. 在 pom.xml 文件中，配置 Maven，以从远程存储库检索 JBoss EAP BOM 文件。

```
<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. 要将 Maven 配置为自动管理 jboss-eap-jakartaee8 BOM 中的 Jakarta EE 工件的版本，请将 BOM 添加到 project pom.xml 文件的 & lt;dependencyManagement > 部分。例如：

```
<dependencyManagement>
 <dependencies>
   <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-jakartaee8</artifactId>
      <version>7.3.4.GA</version>
```

```
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

4. 将由 BOM 管理的 servlet API 工件添加到项目 pom.xml 文件的 < dependency > 部分，如下例所示：

```
<dependency>
    <groupId>org.jboss.spec.javax.servlet</groupId>
    <artifactId>jboss-servlet-api_4.0_spec</artifactId>
    <scope>provided</scope>
</dependency>
```

其他资源

- 有关 JBoss EAP Maven 插件的详情，请参考 JBoss EAP Maven 插件。
- 有关 Galleon 层的详情，请参考为 可引导 JAR 服务器指定 Galleon 层。
- 有关在项目中包含 Red Hat Single Sign-On Galleon 功能软件包的详情，请参考 使用红帽单点登录保护您的 JBoss EAP 可引导 JAR 应用。

8.13. 为您的可引导 JAR 启用 JSON 日志记录

您可以使用 CLI 脚本配置服务器日志记录配置，为可引导 JAR 启用 JSON 日志记录。启用 JSON 日志记录时，您可以使用 JSON 格式查看日志消息。

此流程中的示例演示了如何在裸机平台和 OpenShift 平台上为可引导 JAR 启用 JSON 日志记录。

先决条件

- 您已检查了最新的 Maven 插件版本，如 *MAVEN_PLUGIN_VERSION*.X.GA.Final-redhat-00001，其中 *MAVEN_PLUGIN_VERSION* 是主版本，*X* 是 microversion。请参阅 /ga/org/wildfly/plugins/wildfly-jar-maven-plugin 的索引。
- 您已检查了最新的 Galleon 功能软件包版本，如 4.0.X.GA-redhat-*BUILD_NUMBER*，其中 *X* 是 JBoss EAP XP 的次版本，*BUILD_NUMBER* 是 Galleon 功能软件包的构建号。*X* 和

BUILD_NUMBER 可以在 JBoss EAP XP 4.0.0 产品生命周期中演进。请参阅 /ga/org/jboss/eap/wildfly-galleon-pack 的索引。

- 您已创建了 Maven 项目，设置父依赖项，并添加了用于创建应用程序的依赖项。请参阅 创建可引导 JAR Maven 项目。

重要

在 Maven 项目的 Maven archetype 中，您必须指定特定于项目的 groupID 和 artifactID。例如：

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

注意

流程中显示的示例指定以下属性：

- ${bootable.jar.maven.plugin.version} 用于 Maven 插件版本。
- ${jboss.xp.galleon.feature.pack.version} 用于 Galleon 功能软件包版本。

您必须在项目中设置这些属性。例如：

```
<properties>
  <bootable.jar.maven.plugin.version>6.1.2.Final-redhat-00001</bootable.jar.maven.plugin.version>
  <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-00002</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

1. 将 JBoss Logging 和 Jakarta RESTful Web Services 依赖项（由 BOM 管理）添加到项目

pom.xml 文件的 <dependencies > 部分。例如：

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

2. 在 pom.xml 文件的 <build > 元素中添加以下内容。您必须指定任何 Maven 插件的最新版本，以及 org.jboss.eap:wildfly-galleon-pack Galleon 功能软件包的最新版本。例如：

```
<plugins>
   <plugin>
      <groupId>org.wildfly.plugins</groupId>
      <artifactId>wildfly-jar-maven-plugin</artifactId>
      <version>${bootable.jar.maven.plugin.version}</version>
      <configuration>
         <feature-packs>
            <feature-pack>
               <location>org.jboss.eap:wildfly-galleon-
pack:${jboss.xp.galleon.feature.pack.version}</location>
            </feature-pack>
         </feature-packs>
         <layers>
            <layer>jaxrs-server</layer>
          </layers>
      </configuration>
      <executions>
         <execution>
            <goals>
               <goal>package</goal>
            </goals>
         </execution>
      </executions>
   </plugin>
</plugins>
```

3. 创建用于存储 Java 文件的目录：

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

其中 APPLICATION_ROOT 是包含应用的 pom.xml 配置文件的目录。

4. 创建包含以下内容的 Java 文件 RestApplication.java，并将该文件保存到 APPLICATION_ROOT/src/main/java/com/example/logging/ 目录中：

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

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

5. 创建包含以下内容的 Java 文件 HelloWorldEndpoint.java，并将该文件保存到 APPLICATION_ROOT/src/main/java/com/example/logging/ 目录中：

```
package com.example.logging;

import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
import javax.ws.rs.GET;
import javax.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. 创建一个 CLI 脚本，如 logging.cli，并将它保存到可引导 JAR 的可访问目录中，如 *APPLICATION_ROOT*/scripts 目录，其中 APPLICATION_ROOT 是 Maven 项目的根目录。该脚本必须包含以下命令：

```
/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. 在插件 <configuration> 元素 中添加以下 配置提取：

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

本例演示了 logging.cli CLI 脚本，该脚本修改服务器日志记录配置文件来为应用程序启用 JSON 日志记录。

8. 将应用打包为可引导 JAR。

```
$ mvn package
```

9. *可选* ： 要在 JBoss EAP 裸机平台上运行应用程序，请按照 在 JBoss EAP 裸机平台上使用可引导 JAR 中所述的步骤进行，但有以下区别：

   a. 启动应用程序：

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

   b. 验证 ：您可以通过在浏览器中指定以下 URL 来访问应用程序：
   http://127.0.0.1:8080/hello

   预期的输出：您可以在应用程序控制台中查看 JSON 格式的日志，包括 com.example.logging.HelloWorldEndpoint debug trace。

10. *可选* ： 要在 JBoss EAP OpenShift 平台上运行应用程序，请完成以下步骤：

    a. 在插件配置中添加 <cloud /> 元素。例如：

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

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

b. **重建应用程序：**

```
$ mvn clean package
```

c. **使用 oc login 命令登录到您的 OpenShift 实例。**

d. **在 OpenShift 中创建一个新项目。例如：**

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

e. **输入以下 oc 命令来创建应用程序镜像：**

```
$ mkdir target/openshift && cp target/logging-bootable.jar target/openshift (1)
$ oc import-image ubi8/openjdk-11 --from=registry.redhat.io/ubi8/openjdk-11 --confirm (2)
$ oc new-build --strategy source --binary --image-stream openjdk-11 --name logging (3)
$ oc start-build logging --from-dir target/openshift (4)
```

(1) **创建 target/openshift 子目录。打包的应用程序被复制到 openshift 子目录中。**

(2) **将最新的 OpenJDK 11 镜像流标签和镜像信息导入到 OpenShift 项目中。**

(3) **根据日志记录目录和 OpenJDK 11 镜像流创建构建配置。**

(4)

使用 target/openshift 子目录作为二进制输入来构建应用。

f. 部署应用程序：

```
$ oc new-app logging
$ oc expose svc/logging
```

g. 获取路由的 URL。

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

h. 使用上一命令返回的 URL，访问 Web 浏览器中的应用。例如：

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

i. *验证 ：* 发出以下命令来查看可用的 OpenShift pod 列表，并检查 pod 构建状态：

```
$ oc get pods
```

访问应用程序的正在运行的 pod 日志。其中 APP_POD_NAME 是正在运行的 Pod 日志记录应用的名称。

```
$ oc logs APP_POD_NAME
```

预期的结果：pod 日志采用 JSON 格式，并包含 com.example.logging.HelloWorldEndpoint debug trace。

其他资源

- 有关 JBoss EAP 日志记录功能的详情，请参考 *配置指南中的 使用 JBoss EAP 进行日志记录。*
- 有关在 OpenShift 上使用可引导 JAR 的详情，请参考在 JBoss EAP OpenShift 平台上使用可引导 JAR。

- 有关为项目指定 JBoss EAP JAR Maven 的详情，请参考为 可引导 JAR 服务器指定 Galleon 层。

- 有关创建 CLI 脚本的详情，请参考 CLI 脚本。

8.14. 为多个可引导 JAR 实例启用 WEB 会话数据存储

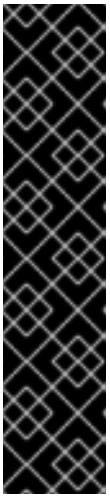
您可以将 web-clustering 应用程序构建并打包为可引导 JAR。

先决条件

- 您已检查了最新的 Maven 插件版本，如 *MAVEN_PLUGIN_VERSION*.X.GA.Final-redhat-00001，其中 *MAVEN_PLUGIN_VERSION* 是主版本，*X* 是 microversion。请参阅 /ga/org/wildfly/plugins/wildfly-jar-maven-plugin 的索引。

- 您已检查了最新的 Galleon 功能软件包版本，如 4.0.X.GA-redhat-*BUILD_NUMBER*，其中 *X* 是 JBoss EAP XP 的微版本，*BUILD_NUMBER* 是 Galleon 功能软件包的构建号。*X* 和 *BUILD_NUMBER* 可以在 JBoss EAP XP 4.0.0 产品生命周期中演进。请参阅 /ga/org/jboss/eap/wildfly-galleon-pack 的索引。

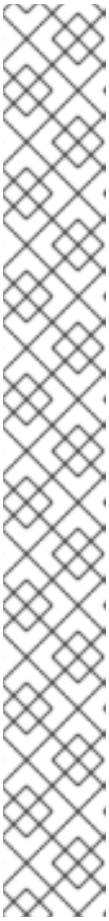
- 您已创建了 Maven 项目，设置父依赖项，并添加了用于创建 Web 集群应用程序的依赖项。请参阅 创建可引导 JAR Maven 项目。



重要

在设置 Maven 项目时，您必须在 Maven archetype 配置中指定值。例如：

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



注意

流程中显示的示例指定以下属性：

- **${bootable.jar.maven.plugin.version} 用于 Maven 插件版本。**
- **${jboss.xp.galleon.feature.pack.version} 用于 Galleon 功能软件包版本。**

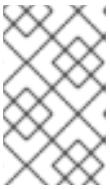
您必须在项目中设置这些属性。例如：

```
<properties>
  <bootable.jar.maven.plugin.version>6.1.2.Final-redhat-
00001</bootable.jar.maven.plugin.version>
  <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-
00002</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

1. **在 pom.xml 文件的 <build > 元素中添加以下内容。您必须指定任何 Maven 插件的最新版本，以及 org.jboss.eap:wildfly-galleon-pack Galleon 功能软件包的最新版本。例如：**

```
<plugins>
  <plugin>
    <groupId>org.wildfly.plugins</groupId>
    <artifactId>wildfly-jar-maven-plugin</artifactId>
    <version>${bootable.jar.maven.plugin.version}</version>
    <configuration>
      <feature-pack-location>org.jboss.eap:wildfly-galleon-
pack:${jboss.xp.galleon.feature.pack.version}</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>
```



注意

这个示例使用 web-clustering Galleon 层启用 Web 会话共享。

2. 使用以下配置更新 src/main/webapp/WEB-INF 目录中的 web.xml 文件：

```
<?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>
```

& lt;distributable /> 标签表示此 servlet 可以在多个服务器间分布。

3. 创建用于存储 Java 文件的目录：

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

其中 APPLICATION_ROOT 是包含应用的 pom.xml 配置文件的目录。

4. 创建包含以下内容的 Java 文件 MyServlet.java，并将该文件保存到 *APPLICATION_ROOT*/src/main/java/com/example/webclustering/ 目录中。

```
package com.example.webclustering;

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

@WebServlet(urlPatterns = {"/clustering"})
public class MyServlet extends HttpServlet {
   @Override
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws IOException {
```

```
        response.setContentType("text/html;charset=UTF-8");
        long t;
        User user = (User) request.getSession().getAttribute("user");
        if (user == null) {
            t = System.currentTimeMillis();
            user = new User(t);
            request.getSession().setAttribute("user", user);
        }
        try (PrintWriter out = response.getWriter()) {
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Web clustering demo</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Session id " + request.getSession().getId() + "</h1>");
            out.println("<h1>User Created " + user.getCreated() + "</h1>");
            out.println("<h1>Host Name " + System.getenv("HOSTNAME") + "</h1>");
            out.println("</body>");
            out.println("</html>");
        }
    }
}
```

MyServlet.java 中的内容定义客户端向其发送 HTTP 请求的端点。

5. 创建包含以下内容的 Java 文件 User.java，并将文件保存到 *APPLICATION_ROOT*/src/main/java/com/example/webclustering/ 目录中。

```
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. 打包应用程序：

```
$ mvn package
```

7.

可选 ： 要在 JBoss EAP 裸机平台上运行应用程序，请按照 在 JBoss EAP 裸机平台上使用可引导 JAR 中所述的步骤进行，但有以下区别：

a.

在 JBoss EAP 裸机平台上，您可以使用 java -jar 命令运行多个可引导 JAR 实例，如下例所示：

```
$ 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.

验证 ：您可以访问节点 1 实例中的应用程序： http://127.0.0.1:8080/clustering。记录用户会话 ID 和用户创建时间。

终止此实例后，您可以访问节点 2 实例： http://127.0.0.1:8090/clustering。用户必须与会话 ID 和节点 1 实例的用户创建时间匹配。

8.

可选 ： 要在 JBoss EAP OpenShift 平台上运行应用程序，请按照 在 JBoss EAP OpenShift 平台上使用可引导 JAR 中所述的步骤，但完成以下步骤：

a.

在插件配置中添加 <cloud /> 元素。例如：

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

b.

重建应用程序：

```
$ mvn clean package
```

c.

使用 oc login 命令登录到您的 OpenShift 实例。

d. 在 OpenShift 中创建一个新项目。例如：

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

e. 若要在 JBoss EAP OpenShift 平台上运行 web-clustering 应用，必须为 Pod 在其中运行的服务帐户授予授权访问权限。然后，服务帐户可以访问 Kubernetes REST API。以下示例显示了为服务帐户授予授权访问权限：

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

f. 输入以下 oc 命令来创建应用程序镜像：

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

$ oc import-image ubi8/openjdk-11 --from=registry.redhat.io/ubi8/openjdk-11 --confirm (2)

$ oc new-build --strategy source --binary --image-stream openjdk-11 --name web-clustering (3)

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

(1) 创建 target/openshift 子目录。打包的应用程序被复制到 openshift 子目录。

(2) 将最新的 OpenJDK 11 镜像流标签和镜像信息导入到 OpenShift 项目中。

(3) 根据 web-clustering 目录和 OpenJDK 11 镜像流创建构建配置。

(4) 使用 target/openshift 子目录作为二进制输入来构建应用程序。

g. 部署应用程序：

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

$ oc expose svc/web-clustering
```

重要

您必须使用 KUBERNETES_NAMESPACE 环境变量查看当前 OpenShift 命名空间中的其他 pod；否则，服务器会尝试从 default 命名空间检索 pod。

h. **获取路由的 URL。**

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

i. **使用上一命令返回的 URL，访问 Web 浏览器中的应用。例如：**

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

记录用户会话 ID 和用户创建时间。

j. **将应用程序扩展到两个 pod：**

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

k. **发出以下命令来查看可用的 OpenShift pod 列表，并检查 pod 构建状态：**

```
$ oc get pods
```

l. **使用 oc delete pod web-clustering-*POD_NAME*命令终止 最旧的 pod，其中 *POD_NAME* 是最旧的 pod 的名称。**

m. **再次访问应用程序：**

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

预期的结果：会话 ID 和新 pod 生成的创建时间与终止的 pod 的创建时间匹配。这表示

启用了 Web 会话数据存储。

其他资源

- 有关可分布式 Web 会话管理配置集的详情，请参考 *开发指南中的* 可分布式 Web 会话配置的 distribut-web 子系统。

- 有关配置 JGroups 协议堆栈的详情，请参考 *OpenShift Container Platform 入门指南* 中的配置 JGroups Discovery Mechanism。

8.15. 使用 CLI 脚本为可引导 JAR 启用 HTTP 身份验证

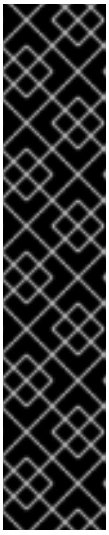
您可以使用 CLI 脚本为可引导 JAR 启用 HTTP 身份验证。此脚本会在您的服务器中添加安全域和一个安全域。

先决条件

- 您已检查了最新的 Maven 插件版本，如 *MAVEN_PLUGIN_VERSION*.X.GA.Final-redhat-00001，其中 *MAVEN_PLUGIN_VERSION* 是主版本，*X* 是 microversion。请参阅 /ga/org/wildfly/plugins/wildfly-jar-maven-plugin 的索引。

- 您已检查了最新的 Galleon 功能软件包版本，如 4.0.X.GA-redhat-*BUILD_NUMBER*，其中 *X* 是 JBoss EAP XP 的微版本，*BUILD_NUMBER* 是 Galleon 功能软件包的构建号。*X* 和 *BUILD_NUMBER* 可以在 JBoss EAP XP 4.0.0 产品生命周期中演进。请参阅 /ga/org/jboss/eap/wildfly-galleon-pack 的索引。

- 您已创建了 Maven 项目，设置父依赖项，并添加了创建需要 HTTP 身份验证的应用程序的依赖项。请参阅 创建可引导 JAR Maven 项目。



重要

在设置 Maven 项目时，您必须在 Maven archetype 配置中指定 HTTP 身份验证值。例如：

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

注意

流程中显示的示例指定以下属性：

- ${bootable.jar.maven.plugin.version} 用于 Maven 插件版本。
- ${jboss.xp.galleon.feature.pack.version} 用于 Galleon 功能软件包版本。

您必须在项目中设置这些属性。例如：

```
<properties>
  <bootable.jar.maven.plugin.version>6.1.2.Final-redhat-
00001</bootable.jar.maven.plugin.version>
  <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-
00002</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

1. 在 pom.xml 文件的 <build > 元素中添加以下内容。您必须指定任何 Maven 插件的最新版本，以及 org.jboss.eap:wildfly-galleon-pack Galleon 功能软件包的最新版本。例如：

```
<plugins>
  <plugin>
    <groupId>org.wildfly.plugins</groupId>
    <artifactId>wildfly-jar-maven-plugin</artifactId>
    <version>${bootable.jar.maven.plugin.version}</version>
    <configuration>
```

```
            <feature-pack-location>org.jboss.eap:wildfly-galleon-
pack:${jboss.xp.galleon.feature.pack.version}</feature-pack-location>
            <layers>
                <layer>datasources-web-server</layer>
            </layers>
        </configuration>
        <executions>
            <execution>
                <goals>
                    <goal>package</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
```

示例显示包含包含 elytron 子系统的 datasources-web-server Galleon 层。

2. 更新 src/main/webapp/WEB-INF 目录中的 web.xml 文件。例如：

```
<?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. 创建用于存储 Java 文件的目录：

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

其中 APPLICATION_ROOT 是 Maven 项目的根目录。

4. 创建包含以下内容的 Java 文件 TestServlet.java，并将该文件保存到 APPLICATION_ROOT/src/main/java/com/example/authentication/ 目录中。

```
package com.example.authentication;
```

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

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

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

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

}
```

5. 创建一个 CLI 脚本，如 authentication.cli，并将它保存到可引导 JAR 的可访问目录中，如 *APPLICATION_ROOT*/scripts 目录。该脚本必须包含以下命令：

```
/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. 在插件 <configuration> 元素 中添加以下 配置提取：

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

本例演示了 authentication.cli CLI 脚本，它将默认 undertow 安全域配置为为您的服务器定

义的安全域。

注意

您可以选择在运行时执行 CLI 脚本，而不是打包时间。为此，请跳过这一步，然后继续下一步 10。

7. 在 Maven 项目的根目录中，创建一个目录来存储 JBoss EAP JAR Maven 插件添加到可引导 JAR 中的属性文件：

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

其中 APPLICATION_ROOT 是包含应用的 pom.xml 配置文件的目录。

此目录存储文件，如 bootable-users.properties 和 bootable-groups.properties 文件。

bootable-users.properties 文件包含以下内容：

```
testuser=bootable_password
```

bootable-groups.properties 文件包含以下内容：

```
testuser=Users
```

8. 将以下 extra-content-content-dirs 元素添加到现有 < configuration> 元素中：

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

extra-content 目录包含属性文件。

9. 将应用打包为可引导 JAR。

```
$ mvn package
```

10. 启动应用程序：

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

如果您选择了跳过第 6 步，且在构建期间没有执行 CLI 脚本，使用以下命令启动应用程序：

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

11. 调用 servlet，但不指定凭证：

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

预期输出：

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

12. 调用服务器并指定您的凭证。例如：

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

返回 HTTP 200 状态，表示为可引导 JAR 启用了 HTTP 身份验证。例如：

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

其他资源

- 有关为 undertow 安全域启用 HTTP 身份验证的详情，请参考 *如何配置服务器安全性 中的 使用 CLI 安全命令为应用程序启用 HTTP 身份验证*。

8.16. 使用红帽单点登录保护您的 JBOSS EAP 可引导 JAR 应用

您可以使用 Galleon keycloak-client-oidc 层安装使用 Red Hat Single Sign-On 7.5 OpenID Connect 客户端适配器置备的服务器版本。

注意

使用 keycloak-client-oidc 层已在 JBoss EAP XP 4 中弃用。使用 elytron-oidc-client 层，它提供原生 OpenID Connect (OIDC)客户端。如需更多信息，请参阅使用 OpenID Connect 开发 JBoss EAP bootable jar 应用。

keycloak-client-oidc 层向 Maven 项目提供 Red Hat Single Sign-On OpenID Connect 客户端适配器。此层包含在 keycloak-adapter-galleon-pack Red Hat Single Sign-On 功能 pack 中。

您可以将 keycloak-adapter-galleon-pack 功能 pack 添加到 JBoss EAP Maven 插件配置中，然后添加 keycloak-client-oidc。您可以通过访问 支持的配置： Red Hat Single Sign-On 7.4 网页来查看与 JBoss EAP 兼容的 Red Hat Single Sign-On 客户端适配器。

此流程中的示例演示了如何使用 keycloak-client-oidc 层提供的 JBoss EAP 功能来保护 JBoss EAP 的可引导 JAR。

先决条件

- 您已检查了最新的 Maven 插件版本，如 *MAVEN_PLUGIN_VERSION*.X.GA.Final-redhat-00001，其中 *MAVEN_PLUGIN_VERSION* 是主版本，*X* 是 microversion。请参阅 /ga/org/wildfly/plugins/wildfly-jar-maven-plugin 的索引。
- 您已检查了最新的 Galleon 功能软件包版本，如 4.0.X.GA-redhat-*BUILD_NUMBER*，其中 *X* 是 JBoss EAP XP 的微版本，*BUILD_NUMBER* 是 Galleon 功能软件包的构建号。*X* 和 *BUILD_NUMBER* 可以在 JBoss EAP XP 4.0.0 产品生命周期中演进。请参阅 /ga/org/jboss/eap/wildfly-galleon-pack 的索引。
- 您已检查了最新的 Red Hat Single Sign-On Galleon 功能软件包版本，如 org.keycloak:keycloak-adapter-galleon-pack:15.0.X.redhat-BUILD_NUMBER，其中 X 是 Red Hat Single Sign-On 的微版本，它依赖于 Red Hat Single Sign-On 服务器版本来保护应用程序，BUILD_NUMBER 是 Red Hat Single Sign-On Galleon 功能的构建号。*X* 和 *BUILD_NUMBER* 可以在 JBoss EAP XP 4.0.0 产品生命周期中演进。请参阅 /ga/org/keycloak/keycloak-adapter-galleon-pack 的索引。
- 您已创建了 Maven 项目，设置父依赖项并添加依赖项，以创建您希望使用 Red Hat Single Sign-On 保护的应用程序。请参阅 创建可引导 JAR Maven 项目。

- 您有一个在端口 8090 上运行的 Red Hat Single Sign-On 服务器。请参阅 启动 Red Hat Single Sign-On 服务器。

- 您已登录到 Red Hat Single Sign-On Admin 控制台并创建以下元数据：

  - 名为 demo 的域。

  - 名为 Users 的角色。

  - 用户和密码。您必须为用户分配 Users 角色。

  - 具有根 URL 的公共客户端 Web 应用。该流程中的示例，将 simple-webapp 定义为 Web 应用程序， http://localhost:8080/simple-webapp/secured 作为根 URL。

重要

在设置 Maven 项目时，您必须在 Maven archetype 中使用 Red Hat Single Sign-On 保护的应用程序指定值。例如：

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

注意

流程中显示的示例指定以下属性：

- ${bootable.jar.maven.plugin.version} 用于 Maven 插件版本。
- ${jboss.xp.galleon.feature.pack.version} 用于 Galleon 功能软件包版本。
- ${Keycloak.feature.pack.version} 用于 Red Hat Single Sign-On feature pack 版本。

您必须在项目中设置这些属性。例如：

```
<properties>
    <bootable.jar.maven.plugin.version>6.1.2.Final-redhat-
00001</bootable.jar.maven.plugin.version>
    <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-
00002</jboss.xp.galleon.feature.pack.version>
    <keycloak.feature.pack.version>15.0.4.redhat-
00001</keycloak.feature.pack.version>
</properties>
```

流程

1. 在 pom.xml 文件的 <build > 元素中添加以下内容。您必须指定任何 Maven 插件的最新版本，以及 org.jboss.eap:wildfly-galleon-pack Galleon 功能软件包的最新版本。例如：

```
<plugins>
    <plugin>
        <groupId>org.wildfly.plugins</groupId>
        <artifactId>wildfly-jar-maven-plugin</artifactId>
        <version>${bootable.jar.maven.plugin.version}</version>
        <configuration>
            <feature-packs>
                <feature-pack>
                    <location>org.jboss.eap:wildfly-galleon-
pack:${jboss.xp.galleon.feature.pack.version}</location>
                </feature-pack>
                <feature-pack>
                    <location>org.keycloak:keycloak-adapter-galleon-
pack:${keycloak.feature.pack.version}</location>
                </feature-pack>
```

```
            </feature-packs>
            <layers>
               <layer>datasources-web-server</layer>
               <layer>keycloak-client-oidc</layer>
            </layers>
         </configuration>
         <executions>
            <execution>
               <goals>
                  <goal>package</goal>
               </goals>
            </execution>
         </executions>
      </plugin>
   </plugins>
```

Maven 插件调配部署 Web 应用所需的子系统和模块。

keycloak-client-oidc 层使用 keycloak 子系统及其依赖项向项目提供 Red Hat Single Sign-On OpenID Connect 客户端适配器，以激活对 Red Hat Single Sign-On 身份验证的支持。Red Hat Single Sign-On 客户端适配器是使用 Red Hat Single Sign-On 保护应用程序和服务的库。

2. **在项目 pom.xml 文件中，在插件配置中将 <context-root > 设置为 false。这会将应用程序注册到 simple-webapp 资源路径中。默认情况下，WAR 文件在 root-context 路径下注册。**

```
<configuration>
   ...
    <context-root>false</context-root>
    ...
</configuration>
```

3. **创建一个 CLI 脚本，如 configure-oidc.cli，并将它保存到可引导 JAR 的可访问目录中，如 *APPLICATION_ROOT*/scripts 目录，其中 *APPLICATION_ROOT* 是 Maven 项目的根目录。该脚本必须包含类似以下示例的命令：**

```
/subsystem=keycloak/secure-deployment=simple-webapp.war:add( \
   realm=demo, \
   resource=simple-webapp, \
   public-client=true, \
   auth-server-url=http://localhost:8090/auth/, \
   ssl-required=EXTERNAL)
```

此脚本示例在 keycloak 子系统中定义 secure-deployment=simple-webapp.war 资源。simple-webapp.war 资源是在可引导 JAR 中部署的 WAR 文件的名称。

4. 在项目 pom.xml 文件中，将以下配置提取添加到现有插件 < configuration> 元素中：

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

5. 更新 src/main/webapp/WEB-INF 目录中的 web.xml 文件。例如：

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

<web-app version="4.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_4_0.xsd"
  metadata-complete="false">

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

</web-app>
```

6. *可选：* 通过添加 keycloak.json 描述符到 web 应用程序的 WEB-INF 目录中，您还可以将服务器配置嵌入到 web 应用中。例如：

```
{
   "realm" : "demo",
   "resource" : "simple-webapp",
   "public-client" : "true",
   "auth-server-url" : "http://localhost:8090/auth/",
   "ssl-required" : "EXTERNAL"
}
```

然后，您必须将 web 应用程序的 <auth-method > 设置为 KEYCLOAK。以下示例代码演示了如何设置 < auth-method> ：

```
   <login-config>
      <auth-method>KEYCLOAK</auth-method>
      <realm-name>Simple Realm</realm-name>
   </login-config>
```

7. 创建名为 SecuredServlet.java 的 Java 文件，其中包含以下内容，并将文件保存到 *APPLICATION_ROOT*/src/main/java/com/example/securedservlet/ 目录中。

```
package com.example.securedservlet;

import java.io.IOException;
import java.io.PrintWriter;
import java.security.Principal;

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

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

   @Override
   protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
      try (PrintWriter writer = resp.getWriter()) {
         writer.println("<html>");
         writer.println("<head><title>Secured Servlet</title></head>");
         writer.println("<body>");
         writer.println("<h1>Secured Servlet</h1>");
         writer.println("<p>");
         writer.print(" Current Principal '");
         Principal user = req.getUserPrincipal();
         writer.print(user != null ? user.getName() : "NO AUTHENTICATED USER");
         writer.print("'");
         writer.println("    </p>");
         writer.println("  </body>");
         writer.println("</html>");
      }
   }
}
```

8. 将应用打包为可引导 JAR。

```
$ mvn package
```

9. 启动应用程序。以下示例从指定的可引导 JAR 路径启动 simple-webapp Web 应用程序：

```
$ java -jar target/simple-webapp-bootable.jar
```

10. 在 Web 浏览器中指定以下 URL，以访问通过 Red Hat Single Sign-On 保护的网页。以下示例显示了安全 simple-webapp Web 应用程序的 URL：

```
http://localhost:8080/simple-webapp/secured
```

11. 以来自您的 Red Hat Single Sign-On 域的用户身份登录。

12. *验证*：检查网页是否显示以下输出：

```
Current Principal '<principal id>'
```

其他资源

- 有关配置 Red Hat Single Sign-On adapter 子系统的详情，请参考 *保护应用程序和服务指南中的* JBoss EAP Adapter。

- 有关为项目指定 JBoss EAP JAR Maven 的详情，请参考为 可引导 JAR 服务器指定 Galleon 层。

8.17. 在 DEV 模式中打包可引导 JAR

JBoss EAP JAR Maven 插件 dev 目标 提供 dev 模式开发模式，可用于增强应用程序开发流程。

在 dev 模式中，在对应用进行更改后，您不需要重建可引导 JAR。

此流程中的工作流演示了使用 dev 模式配置可引导 JAR。

先决条件

- 已安装 Maven。

- 您已创建了 Maven 项目，设置父依赖项，并添加了用于创建 MicroProfile 应用的依赖项。请参阅 MicroProfile 配置开发。

- 您已在 Maven 项目 pom.xml 文件中指定了 JBoss EAP JAR Maven 插件。

流程

1. 在开发模式中构建并启动可引导 JAR：

```
$ mvn wildfly-jar:dev
```

在 dev 模式中，服务器部署扫描程序配置为监控 target/deployments 目录。

2. 使用以下命令提示 JBoss EAP Maven 插件构建并复制到 target/deployments 目录中：

```
$ mvn package -Ddev
```

打包在可引导 JAR 中的服务器部署存储在 target/deployments 目录中的应用。

3. 修改应用程序代码中的代码。

4. 使用 mvn package -Ddev 来提示输入 JBoss EAP Maven 插件，以重新构建您的应用程序并重新部署它。

5. 停止服务器。例如：

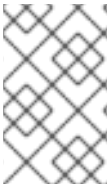
```
$ mvn wildfly-jar:shutdown
```

6. 完成应用程序更改后，将应用程序打包为可引导 JAR:

```
$ mvn package
```

8.18. 升级服务器工件

服务器工件是位于 JBoss 模块中的 jar 文件，您可以使用项目 pom.xml 文件中的 Maven 协调来引用它。



注意

请注意，升级服务器工件可能会导致不支持的配置。

先决条件

- 确保 Maven 工件可从您的本地 Maven 存储库或远程 Maven 存储库访问。

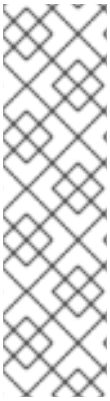
流程

1. 在构建期间使用依赖项中存在的工件版本，以成功升级服务器工件。例如：

```
<dependencies>
  ...
  <dependency>
 <groupId>io.undertow</groupId>
 <artifactId>undertow-core</groupId>
  <version>2.2.5.Final-redhat-00001</version>
        <scope>provided</scope>
        <!-- In order to avoid bringing transitive dependencies to the project, exclude all
dependencies -->
        <exclusions>
          <exclusion>
            <groupId>*</groupId>
            <artifactId>*</artifactId>
          </exclusion>
        </exclusions>
  </dependency>
  ...
</dependencies>
```

2. 打开插件 & lt;configuration& gt; 部分，并在 < overridden-server-artifacts& gt; 列表中更新工件 groupId 和 artifactId，例如：

```
<configuration>
 ...
 <overridden-server-artifacts>
  <artifact>
   <groupId>io.undertow</groupId>
   <artifactId>undertow-core</groupId>
  </artifact>
 </overridden-server-artifacts>
</configuration>
```

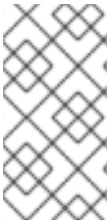


注意

- 如果项目依赖项中没有添加到 <overridden-server-artifacts > 的工件，则会出现失败。
- 如果添加到 &lt ;overridden-server-artifacts& gt; 的工件不在置备的服务器工件中，则会发生失败，因为升级的目标工件无法找到。

8.19. 更新 EAP 7.4.GA 依赖项

为 JBoss EAP XP 4.0.0 构建可引导 JAR 时，您可以更新 JBoss EAP XP 4.0.0 上的依赖项。JBoss EAP XP 4.0.0 galleon feature-pack org.jboss.eap:wildfly-galleon-pack:4.0.0.GA-redhat-00001 依赖于 org.jboss.eap:wildfly-ee-galleon-pack:7.4.0.GA-redhat-00001，您可以在构建 Bootable JAR 时升级。



注意

升级到最新版本的 JBoss EAP XP 版本。这样可确保您在 JBoss EAP XP 4.0.0 可引导 JAR 中获得最新的更新。

先决条件

- 您有最新版本的 JBoss EAP XP。

流程

1. 确保 JBoss EAP Galleon feature-pack Maven 工件可从您的本地或远程 Maven 存储库访问。
2. 在项目依赖项中添加 Galleon 功能包工件：
   a. 将范围设置为 提供的。
   b. 将类型设置为 zip。

c.

设置工件版本。例如：

```
<dependencies>
  ...
  <dependency>
 <groupId>org.jboss.eap</groupId>
 <artifactId>wildfly-ee-galleon-pack</groupId>
  <version>7.4.1.GA-redhat-00001</version>
<scope>provided</scope>
<type>zip</type>
  </dependency>
  ...
</dependencies>
```

3.

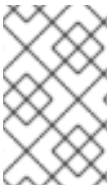
打开插件 & lt;configuration& gt; 部分，并在 < overridden-server-artifacts& gt; 列表中更新工件 groupId 和 artifactId，例如：

```
<configuration>
 ...
 <overridden-server-artifacts>
  <artifact>
   <groupId>org.jboss.eap</groupId>
   <artifactId>wildfly-ee-galleon-pack</groupId>
  </artifact>
 </overridden-server-artifacts>
</configuration>
```

4.

在构建期间使用最新版本的 JBoss EAP XP Galleon 功能包，以成功依赖项。

8.20. 将 JBOSS EAP 补丁应用到您的可引导 JAR



注意

在 JBoss EAP XP 4.0.0 中，用于可引导 jar 的传统补丁功能已弃用。

在 JBoss EAP 裸机平台上，您可以使用 CLI 脚本将补丁安装到可引导 JAR。

CLI 脚本发出 patch apply 命令，以在可引导 JAR 构建期间应用补丁。

重要

将补丁应用到可引导 JAR 后，您无法从应用的补丁中回滚。您必须重建一个可引导 JAR，而无需补丁。

此外，您可以使用 JBoss EAP JAR Maven 插件将传统补丁应用到可引导 JAR。此插件提供了一个 < legacy-patch-cli-script > 配置选项，用于引用用于修补服务器的 CLI 脚本。

注意

&lt ; legacy -patch-cli-script> 中的前缀 legacybang 与将存档补丁应用到可引导 JAR 相关。此方法类似于将补丁应用到常规 JBoss EAP 发行版。

您可以使用 JBoss EAP JAR Maven 插件配置中的 legacy-patch-cleanup 选项，通过删除未使用的补丁内容来减少可引导 JAR 的内存占用。选项会删除未使用的模块依赖项。在补丁配置文件中默认将此选项设置为 false。

legacy-patch-cleanup 选项会删除以下补丁内容：

- & lt;JBOSS_HOME>/.installation/patches 目录。
- 基本层中的补丁模块的原始位置。
- 由补丁添加且没有在现有模块图形或修补的模块图中引用未使用的模块。
- 覆盖没有在 .overlays 文件中列出的目录。

重要

legacy-patch-clean-up 选项变量作为技术预览提供。技术预览功能不包括在红帽生产服务级别协议（SLA）中，且其功能可能并不完善。因此，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

注意

此流程中概述的信息也与 hollow bootable JAR 相关。

先决条件

- 您已 在红帽客户门户网站中 设置了帐户。
- 您已从 产品 下载页面下载了以下文件：
  - JBoss EAP JBoss EAP 7.4.4 GA 补丁
  - JBoss EAP XP 4.0.0 补丁

流程

1. 创建一个 CLI 脚本，用于定义您要应用到可引导 JAR 的传统补丁。该脚本必须包含一个或多个补丁应用命令。在修补使用 Galleon 层修剪的服务器时，需要 --override-all 命令，例如：

```
patch apply patch-oneoff1.zip --override-all

patch apply patch-oneoff2.zip --override-all

patch info --json-output
```

2. 在 pom.xml 文件的 & lt;legacy-patch-cli-script > 元素中引用您的 CLI 脚本。
3. 重建可引导 JAR。

其他资源

- 有关下载 JBoss EAP MicroProfile Maven 存储库的详情，请参考 下载 JBoss EAP MicroProfile Maven 存储库补丁作为存档文件。
- 有关创建 CLI 脚本的详情，请参考 CLI 脚本。

- 有关 技术预览功能的详情，请参考 *红帽客户门户网站中的* 技术预览功能支持范围。

第 9 章 JBOSS EAP 中的 OPENID CONNECT

使用 JBoss EAP 原生 OpenID Connect (OIDC)客户端通过外部 OpenID 供应商保护应用程序。OIDC 是一个身份层，它允许客户端（如 JBoss EAP）根据 OpenID 供应商身份验证验证用户身份。例如，您可以使用 Red Hat Single Sign-On 作为 OpenID 供应商来保护 JBoss EAP 应用程序。

9.1. JBOSS EAP 中的 OPENID CONNECT 配置

当您使用 OpenID 供应商保护应用程序时，您不需要在本地配置任何安全域资源。elytron-oidc-client 子系统在 JBoss EAP 中提供原生 OpenID Connect (OIDC)客户端，以与 OpenID 供应商连接。JBoss EAP 根据您的 OpenID 提供程序配置自动为您的应用程序创建虚拟安全域。

重要

建议您在 Red Hat Single Sign-On 中使用 OIDC 客户端。如果可将其他 OpenID 供应商配置为使用 JSON Web 令牌(JWT)的访问令牌，并可配置为使用 RS256、RS384、RS512、ES256、ES384 或 ES512 签名算法。

要启用 OIDC 的使用，您可以配置 elytron-oidc-client 子系统或应用程序本身。JBoss EAP 激活 OIDC 身份验证，如下所示：

- 当您将应用程序部署到 JBoss EAP 时，elytron-oidc-client 子系统会扫描部署，以检测是否需要 OIDC 身份验证机制。
- 如果子系统在 elytron-oidc-client 子系统或应用程序部署描述符中检测到部署的 OIDC 配置，JBoss EAP 为应用启用 OIDC 身份验证机制。
- 如果子系统在两个位置检测到 OIDC 配置，则 elytron-oidc-client 子系统 secure-deployment 属性中的配置优先于应用程序部署描述符中的配置。

注意

使用 Red Hat Single Sign-On 保护应用程序 keycloak-client-oidc 层已在 JBoss EAP XP 4.0.0 中弃用。使用 elytron-oidc-client 子系统提供的原生 OIDC 客户端。

部署配置

要使用部署描述符使用 OIDC 保护应用程序，请更新应用程序的部署配置，如下所示：

- 在 WEB-INF 目录中创建一个名为 oidc.json 的文件，其中包含 OIDC 配置信息。

oidc.json 内容示例

```
{
 "client-id" : "customer-portal", 1
 "provider-url" : "http://localhost:8180/auth/realms/demo", 2
 "ssl-required" : "external", 3
  "credentials" : {
    "secret" : "234234-234234-234234" 4
  }
}
```

1 使用 OpenID 供应商标识 OIDC 客户端的名称。

2 OpenID 提供程序 URL。

3 对外部请求需要 HTTPS。

4 与 OpenID 供应商注册的客户端 secret。

- 在应用程序部署描述符 web.xml 文件中将 auth-method 属性设置为 OIDC。

部署描述符更新示例

```
<login-config>
    <auth-method>OIDC</auth-method>
</login-config>
```

子系统配置

您可以通过使用以下方法配置 elytron-oidc-client 子系统来保护使用 OIDC 的应用程序：

- 如果您为每个应用程序使用相同的 OpenID 供应商，请为多个部署创建一个单一配置。
- 如果您将不同的 OpenID 供应商用于不同的应用程序，请为每个部署创建不同的配置。

单个部署的 XML 配置示例：

```
<subsystem xmlns="urn:wildfly:elytron-oidc-client:1.0">
    <secure-deployment name="DEPLOYMENT_RUNTIME_NAME.war"> 1
        <client-id>customer-portal</client-id> 2
        <provider-url>http://localhost:8180/auth/realms/demo</provider-url> 3
        <ssl-required>external</ssl-required> 4
        <credential name="secret" secret="0aa31d98-e0aa-404c-b6e0-e771dba1e798" /> 5
    </secure-deployment>
</subsystem>
```

1. 部署运行时名称。
2. 使用 OpenID 供应商标识 OIDC 客户端的名称。
3. OpenID 提供程序 URL。
4. 对外部请求需要 HTTPS。
5.

要使用同一 OpenID 供应商保护多个应用程序，请单独配置 提供程序，如下例所示：

```
<subsystem xmlns="urn:wildfly:elytron-oidc-client:1.0">
    <provider name="${OpenID_provider_name}">
        <provider-url>http://localhost:8080/auth/realms/demo</provider-url>
        <ssl-required>external</ssl-required>
    </provider>
    <secure-deployment name="customer-portal.war"> (1)
        <provider>${OpenID_provider_name}</provider>
        <client-id>customer-portal</client-id>
        <credential name="secret" secret="0aa31d98-e0aa-404c-b6e0-e771dba1e798" />
    </secure-deployment>
    <secure-deployment name="product-portal.war"> (2)
        <provider>${OpenID_provider_name}</provider>
        <client-id>product-portal</client-id>
        <credential name="secret" secret="0aa31d98-e0aa-404c-b6e0-e771dba1e798" />
    </secure-deployment>
</subsystem>
```

(1) **一个部署： customer-portal.war**

(2) **另一个部署： product-portal.war**

其他资源

- OpenID Connect 规格
- Elytron-oidc-client 子系统属性
- OpenID Connect Libraries
- 使用 OpenID Connect 与红帽单点登录保护应用程序
- MicroProfile JWT

9.2. 启用 ELYTRON-OIDC-CLIENT 子系统

elytron-oidc-client 子系统在 standalone-microprofile.xml 配置文件中提供。要使用它，您必须使用 bin/standalone.sh -c standalone-microprofile.xml 命令启动您的服务器。您可以通过在 standalone.xml 配置中包含 elytron-oidc-client 子系统，方法是使用管理 CLI 启用它。

先决条件

- 已安装 JBoss EAP XP。

流程

1. 使用管理 CLI 添加 elytron-oidc-client 扩展。

```
/extension=org.wildfly.extension.elytron-oidc-client:add
```

2. 使用管理 CLI 启用 elytron-oidc-client 子系统。

```
/subsystem=elytron-oidc-client:add
```

3. 重新加载 JBoss EAP。

```
reload
```

现在，您可以使用命令 bin/standalone.sh命令来正常启动服务器来使用 elytron-oidc-client 子系统

其他资源

- Elytron-oidc-client 子系统属性

9.3. 使用 OPENID CONNECT 与红帽单点登录保护应用程序

您可以使用 OpenID Connect (OIDC)将身份验证委派给外部 OpenID 供应商。elytron-oidc-client 子系统在 JBoss EAP 中提供原生 OIDC 客户端，以与外部 OpenID 供应商连接。

要使用 Red Hat Single Sign-On 创建使用 OpenID Connect 保护的应用程序，请按照以下步骤执行：

- 将 Red Hat Single Sign-On 配置为 OpenID 供应商
- 为您的应用程序创建一个 Maven 项目
- 创建使用 OpenID Connect 的应用程序
- 根据用户角色限制对应用程序的访问
- 在 Red Hat Single Sign-On 中创建并分配用户角色

9.3.1. 将 Red Hat Single Sign-On 配置为 OpenID 供应商

Red Hat Single Sign-On 是一个身份和访问管理供应商，用于通过单点登录(SSO)保护 Web 应用程序。它支持 OpenID Connect (OAuth 2.0 的扩展)。

先决条件

- 已安装 Red Hat Single Sign-On 服务器。如需更多信息，请参阅 *Red Hat Single Sign-On 入门指南* 中的安装 Red Hat Single Sign-On 服务器。
- 您已在 Red Hat Single Sign-On 服务器实例中创建用户。如需更多信息，请参阅 Red Hat Single Sign-On *入门指南中的创建用户*。https://access.redhat.com/documentation/zh-cn/red_hat_single_sign-on/7.5/html-single/getting_started_guide/#create-user_

流程

1. 在 8080 之外的端口启动 Red Hat Single Sign-On 服务器，因为 JBoss EAP 默认端口为 8080。

   语法

```
$ RH_SSO_HOME/bin/standalone.sh -Djboss.socket.binding.port-offset=<offset-number>
```

Example

```
$ /home/servers/rh-sso-7.4/bin/standalone.sh -Djboss.socket.binding.port-offset=100
```

2. 登录到位于 **http://localhost:<port>/auth/** 的管理控制台。例如：**http://localhost:8180/auth/**。

3. 要创建域，请在管理门户中，将鼠标悬停在 **Master** 上，然后单击 **Add realm**。

4. 输入 **realm** 的名称。例如： **example_realm**。确保 **Enabled** 为 **ON**，再单击 **Create**。

5. 点 **Users**，然后点 **Add user** 将用户添加到域中。

6. 输入用户名。例如，**jane_doe**。确保 **User Enabled** 为 **ON**，然后单击 **Save**。

7. 点 **Credentials** 为用户添加密码。

8. 设置用户的密码。例如，**janedoep@$$**。将 **Temporary** 切换到 **OFF**，然后单击 **Set Password**。在确认提示中，单击 **Set password**。

9. 点 **Clients**，然后点 **Create** 来配置客户端连接。

10. 输入客户端 **ID**。例如，**my_jbeap**。确保将 **Client Protocol** 设置为 **openid-connect**，然后单击 **Save**。

11. 点 Installation，然后选择 Keycloak OIDC JSON 作为 Format Option 来查看连接参数。

```
{
  "realm": "example_realm",
  "auth-server-url": "http://localhost:8180/auth/",
  "ssl-required": "external",
  "resource": "my_jbeap",
  "public-client": true,
  "confidential-port": 0
}
```

在将 JBoss EAP 应用程序配置为使用 Red Hat Single Sign-On 作为身份提供程序时，您可以使用参数，如下所示：

```
"provider-url" : "http://localhost:8180/auth/realms/example_realm",
"ssl-required": "external",
"client-id": "my_jbeap",
"public-client": true,
"confidential-port": 0
```

12. 点 Clients，点 my_jbeap 旁边的 Edit 来编辑客户端设置。

13. 在 Valid Redirect URI 中，输入身份验证成功后页面应重定向的 URL。

在本例中，将此值设置为 http://localhost:8080/simple-oidc-example/secured/*

其他资源

- 配置 Maven 项目以创建安全应用程序
- 创建域和用户

9.3.2. 配置 Maven 项目以创建安全应用程序

创建一个具有所需依赖项的 Maven 项目，以及用于创建安全应用程序的目录结构。

先决条件

- 您已安装了 Maven。如需更多信息，请参阅 下载 Apache Maven。

- 您已为最新版本配置了 Maven 存储库。如需更多信息，请参阅 Maven 和 JBoss EAP microprofile maven 存储库。

流程

1. 使用 mvn 命令建立一个 Maven 项目。该命令创建项目的目录结构以及 pom.xml 配置文件。

语法

```
$ 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.oidc \
-DartifactId=simple-oidc-example \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-webapp \
-DinteractiveMode=false
```

2. 进入到应用程序根目录：

语法

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

Example

```
$ cd simple-oidc-example
```

3. **更新生成的 pom.xml 文件，如下所示：**

   a. **设置以下属性：**

```
<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.server.bootable-jar>4.0.0.GA</version.server.bootable-jar>
    <version.wildfly-jar.maven.plugin>4.0.0.GA</version.wildfly-jar.maven.plugin>
</properties>
```

   b. **设置以下依赖项：**

```
<dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.1.0.redhat-1</version>
      <scope>provided</scope>
    </dependency>
</dependencies>
```

   c. **将以下构建配置设置为使用 mvn widlfy:deploy 来部署应用程序：**

```
<build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
        <plugin>
```

```
            <groupId>org.wildfly.plugins</groupId>
            <artifactId>wildfly-maven-plugin</artifactId>
            <version>2.1.0.Final</version>
        </plugin>
    </plugins>
</build>
```

验证

- 在应用程序根目录中，输入以下命令：

```
$ mvn install
```

您会看到类似如下的输出：

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.440 s
[INFO] Finished at: 2021-12-27T14:45:12+05:30
[INFO] ------------------------------------------------------------------------
```

现在，您可以创建安全应用程序。

其他资源

- 创建使用 OpenID Connect 的安全应用程序

9.3.3. 创建使用 OpenID Connect 的安全应用程序

您可以通过更新其部署配置或配置 elytron-oidc-client 子系统来保护应用。以下示例演示了创建一个显示已登录用户的 Principal 的 servlet。对于现有应用，仅需要与更新部署配置或 elytron-oidc-client 子系统相关的步骤。

在本例中，Principal 的值来自 OpenID 提供程序的 ID 令牌。默认情况下，Principal 是令牌中的 "sub" 声明的值。您可以在以下之一中指定 ID 令牌中的声明值作为 Principal ：

- elytron-oidc-client 子系统属性 principal-attribute。

- oidc.json 文件。

该流程中的 &lt ;*application_root* > 表示 pom.xml 文件目录。pom.xml 文件包含应用的 Maven 配置。

先决条件

- 您已创建了一个 Maven 项目。如需更多信息，请参阅配置 Maven 项目以创建安全应用程序。
- 您已将 Red Hat Single Sign-On 配置为 OpenID 供应商。如需更多信息，请参阅配置 Red Hat Single Sign-On 作为 OpenID 供应商。
- 您已启用了 elytron-oidc-client 子系统。如需更多信息，请参阅启用 elytron-oidc-client 子系统

流程

1. 创建一个用于存储 Java 文件的目录。

语法

```
$ mkdir -p <application_root>/src/main/java/com/example/oidc
```

Example

```
$ mkdir -p simple-oidc-example/src/main/java/com/example/oidc
```

2. 前往新目录。

语法

```
$ cd <application_root>/src/main/java/com/example/oidc
```

Example

```
$ cd simple-oidc-example/src/main/java/com/example/oidc
```

3. 使用以下内容创建 servlet "SecuredServlet.java" :

```
package com.example.oidc;

import java.io.IOException;
import java.io.PrintWriter;
import java.security.Principal;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * A simple secured HTTP servlet.
 *
 */
@WebServlet("/secured")
public class SecuredServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        try (PrintWriter writer = resp.getWriter()) {
            writer.println("<html>");
            writer.println("  <head><title>Secured Servlet</title></head>");
            writer.println("  <body>");
```

```
            writer.println("    <h1>Secured Servlet</h1>");
            writer.println("    <p>");
            writer.print(" Current Principal '");
            Principal user = req.getUserPrincipal();
            writer.print(user != null ? user.getName() : "NO AUTHENTICATED USER");
            writer.print("'");
            writer.println("    </p>");
            writer.println("  </body>");
            writer.println("</html>");
        }
    }

}
```

4. 在应用的 WEB-INF 目录中的部署描述符 web.xml 文件中添加访问应用程序的安全规则。

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

<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  metadata-complete="false">

   <security-constraint>
      <web-resource-collection>
         <web-resource-name>secured</web-resource-name>
         <url-pattern>/secured</url-pattern>
      </web-resource-collection>
      <auth-constraint>
         <role-name>*</role-name>
      </auth-constraint>
   </security-constraint>

   <security-role>
      <role-name>*</role-name>
   </security-role>
</web-app>
```

5. 要使用 OpenID Connect 保护应用，可以更新部署配置或配置 elytron-oidc-client 子系统。

注意

如果您在部署配置和 elytron-oidc-client 子系统中配置 OpenID Connect，则 elytron-oidc-client 子系统 secure-deployment 属性中的配置优先于应用部署描述符中的配置。

- 更新部署配置：

  i. 在 WEB-INF 目录中创建 oidc.json 文件，如下所示：

```
{
  "provider-url" : "http://localhost:8180/auth/realms/example_realm",
  "ssl-required": "external",
  "client-id": "my_jbeap",
  "public-client": true,
  "confidential-port": 0
}
```

  ii. 使用以下文本更新部署描述符 web.xml 文件，以声明此应用程序使用 OIDC：

```
<login-config>
   <auth-method>OIDC</auth-method>
</login-config>
```

- 配置 elytron-oidc-client 子系统：

  - 要保护应用程序，请使用以下管理 CLI 命令：

```
/subsystem=elytron-oidc-client/secure-deployment=simple-oidc-example.war/:add(client-id=my_jbeap,provider-url=http://localhost:8180/auth/realms/example_realm,public-client=true,ssl-required=external)
```

6. 在应用程序根目录中，使用以下命令编译应用程序：

```
$ mvn package
```

7. 部署应用。

```
$ mvn wildfly:deploy
```

验证

1. 在一个浏览器中，导航到 http://localhost:8080/simple-oidc-example/secured。

2. 使用您的凭证登录。例如：

```
username: jane_doe
password: janedoep@$$
```

您将获得以下输出：

```
Secured Servlet
Current Principal '5cb0c4ca-0477-44c3-bdef-04db04d7e39d'
```

现在，您可以使用您在 Red Hat Single Sign-On 中配置的凭证作为 OpenID 供应商登录到应用程序。

其他资源

- JBoss EAP 中的 OpenID Connect 配置
- 根据用户角色限制对应用程序的访问

9.3.4. 根据用户角色限制对应用程序的访问

您可以根据用户角色限制对应用程序的所有或部分的访问。例如，您可以让具有"public"角色的用户能够访问您应用程序的部分不敏感，并授予用户对那些部分的"admin"角色访问权限。

先决条件

- 您已使用 OpenID Connect 保护应用程序。如需更多信息，请参阅创建使用 OpenID Connect 的安全应用程序。

流程

1. 使用以下文本更新部署描述符 web.xml 文件：

语法

```
<security-constraint>
    ...
    <auth-constraint>
        <role-name><allowed_role></role-name>
    </auth-constraint>
</security-constraint>
```

Example

```
<security-constraint>
    ...
    <auth-constraint>
        <role-name>example_role</role-name> (1)
    </auth-constraint>
</security-constraint>
```

(1) 仅允许具有 **example_role** 角色的用户访问您的应用。

2. 在应用程序根目录中，使用以下命令重新编译应用程序：

```
$ mvn package
```

3. 部署应用。

```
$ mvn wildfly:deploy
```

验证

1. 在一个浏览器中，导航到 **http://localhost:8080/simple-oidc-example/secured**。

2. 使用您的凭证登录。例如：

```
username: jane_doe
password: janedoep@$$
```

您将获得以下输出：

```
Forbidden
```

由于您尚未将所需的角色分配给用户 "jane_doe，" jane_doe 无法登录到您的应用程序。只有具有所需角色的用户才能登录。

要为用户分配所需的角色，请参阅 在 Red Hat Single Sign-On 中创建和分配角色。

9.3.5. 在 Red Hat Single Sign-On 中创建并分配用户角色

Red Hat Single Sign-On 是一个身份和访问管理供应商，用于通过单点登录(SSO)保护您的 Web 应用程序。您可以在 Red Hat Single Sign-On 中定义用户并分配角色。

先决条件

- 您已配置了 Red Hat Single Sign-On。如需更多信息，请参阅配置 Red Hat Single Sign-On 作为 OpenID 供应商。

流程

1. 登录到位于 http://localhost:<port>/auth/ 的 admin 控制台。例如：http://localhost:8180/auth/。
2. 单击用于与 JBoss EAP 连接的域。例如： *example_realm*。
3. 单击 Clients，然后单击您为 JBoss EAP 配置的 client-name。例如，*my_jbeap*。
4. 单击 Roles，然后单击 Add Role。
5. 输入角色名称，如 *example_role*，然后单击 Save。这是您在 JBoss EAP 中配置用于授权的角色名称。

6. 单击 **Users**，然后查看所有用户。

7. 点 **ID** 来分配您创建的角色。例如，单击 ***jane_doe*** 的 **ID**。

8. 点 **Role Mappings**。在 **Client Roles** 字段中，选择您为 **JBoss EAP** 配置的 **client-name**。例如，***my_jbeap***。

9. 在 **Available Roles** 中，选择要分配的角色。例如，***example_role***。单击 **Add selected**。

验证

1. 在浏览器中，导航到应用 **URL**。

2. 使用您的凭证登录。例如：

```
username: jane_doe
password: janedoep@$$
```

您将获得以下输出：

```
Secured Servlet
Current Principal '5cb0c4ca-0477-44c3-bdef-04db04d7e39d'
```

具有所需角色的用户可以登录到您的应用程序。

其他资源

- 使用红帽单点登录中的角色和组分配权限和访问权限

9.4. 使用 OPENID CONNECT 开发 JBOSS EAP BOOTABLE JAR 应用程序

您可以使用 OpenID Connect (OIDC)将身份验证委派给外部 OpenID 供应商。elytron-oidc-client galleon 层在 JBoss EAP bootable jar 应用程序中提供原生 OIDC 客户端，以与外部 OpenID 供应商连接。

要使用 Red Hat Single Sign-On 创建使用 OpenID Connect 保护的应用程序，请按照以下步骤执行：

- 将 Red Hat Single Sign-On 配置为 OpenID 供应商
- 为您的应用程序创建一个 Maven 项目
- 创建使用 OpenID Connect 的可引导 jar 应用程序
- 根据用户角色限制对应用程序的访问
- 在 Red Hat Single Sign-On 中创建并分配用户角色

9.4.1. 将 Red Hat Single Sign-On 配置为 OpenID 供应商

Red Hat Single Sign-On 是一个身份和访问管理供应商，用于通过单点登录(SSO)保护 Web 应用程序。它支持 OpenID Connect (OAuth 2.0 的扩展)。

先决条件

- 已安装 Red Hat Single Sign-On 服务器。如需更多信息，请参阅 *Red Hat Single Sign-On 入门指南* 中的安装 Red Hat Single Sign-On 服务器。
- 您已在 Red Hat Single Sign-On 服务器实例中创建用户。如需更多信息，请参阅 Red Hat Single Sign-On *入门指南中的创建用户*。https://access.redhat.com/documentation/zh-cn/red_hat_single_sign-on/7.5/html-single/getting_started_guide/#create-user_

流程

1. 在 8080 之外的端口启动 Red Hat Single Sign-On 服务器，因为 JBoss EAP 默认端口为 8080。

   语法

```
$ RH_SSO_HOME/bin/standalone.sh -Djboss.socket.binding.port-offset=<offset-number>
```

Example

```
$ /home/servers/rh-sso-7.4/bin/standalone.sh -Djboss.socket.binding.port-offset=100
```

2. 登录到位于 **http://localhost:<port>/auth/** 的管理控制台。例如：**http://localhost:8180/auth/**。

3. 要创建域，请在管理门户中，将鼠标悬停在 **Master** 上，然后单击 **Add realm**。

4. 输入 **realm** 的名称。例如： **example_realm**。确保 **Enabled** 为 **ON**，再单击 **Create**。

5. 点 **Users**，然后点 **Add user** 将用户添加到域中。

6. 输入用户名。例如，**jane_doe**。确保 **User Enabled** 为 **ON**，然后单击 **Save**。

7. 点 **Credentials** 为用户添加密码。

8. 设置用户的密码。例如，**janedoep@$$**。将 **Temporary** 切换到 **OFF**，然后单击 **Set Password**。在确认提示中，单击 **Set password**。

9. 点 **Clients**，然后点 **Create** 来配置客户端连接。

10. 输入客户端 **ID**。例如，**my_jbeap**。确保将 **Client Protocol** 设置为 **openid-connect**，然后单击 **Save**。

11. 点 Installation，然后选择 Keycloak OIDC JSON 作为 Format Option 来查看连接参数。

```
{
  "realm": "example_realm",
  "auth-server-url": "http://localhost:8180/auth/",
  "ssl-required": "external",
  "resource": "my_jbeap",
  "public-client": true,
  "confidential-port": 0
}
```

在将 JBoss EAP 应用程序配置为使用 Red Hat Single Sign-On 作为身份提供程序时，您可以使用参数，如下所示：

```
"provider-url" : "http://localhost:8180/auth/realms/example_realm",
"ssl-required": "external",
"client-id": "my_jbeap",
"public-client": true,
"confidential-port": 0
```

12. 点 Clients，点 my_jbeap 旁边的 Edit 来编辑客户端设置。

13. 在 Valid Redirect URI 中，输入身份验证成功后页面应重定向的 URL。

在本例中，将此值设置为 http://localhost:8080/simple-oidc-layer-example/secured/*

其他资源

- 配置 Maven 项目以创建安全应用程序
- 创建域和用户

9.4.2. 为可引导 jar OIDC 应用程序配置 Maven 项目

创建一个具有所需依赖项的 Maven 项目，以及用于创建使用 OpenID Connect 的可引导 jar 应用程序的目录结构。elytron-oidc-client galleon 层提供了一个原生 OpenID Connect (OIDC)客户端来与 OpenID 供应商连接。

先决条件

- 您已安装了 Maven。如需更多信息，请参阅 下载 Apache Maven。

- 您已为最新版本配置了 Maven 存储库。如需更多信息，请参阅 Maven 和 JBoss EAP microprofile Maven 存储库。

流程

1. 使用 mvn 命令建立一个 Maven 项目。该命令创建项目的目录结构以及 pom.xml 配置文件。

语法

```
$ 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.oidc \
-DartifactId=simple-oidc-layer-example \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-webapp \
-DinteractiveMode=false
```

2. 导航到应用程序根目录。

语法

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

Example

```
$ cd simple-oidc-layer-example
```

3. **更新生成的 pom.xml 文件，如下所示：**

   a. **设置以下软件仓库：**

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

   b. **设置以下插件存储库：**

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

   c. **设置以下属性：**

```
<properties>
   <maven.compiler.source>1.8</maven.compiler.source>
```

```
    <maven.compiler.target>1.8</maven.compiler.target>
    <bootable.jar.maven.plugin.version>6.1.2.Final-redhat-
00001</bootable.jar.maven.plugin.version>
    <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-
00002</jboss.xp.galleon.feature.pack.version>
</properties>
```

d. 设置以下依赖项：

```
<dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.1.0.redhat-1</version>
      <scope>provided</scope>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jboss.bom</groupId>
            <artifactId>jboss-eap-jakartaee8</artifactId>
            <version>7.3.4.GA</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.jboss.spec.javax.servlet</groupId>
            <artifactId>jboss-servlet-api_4.0_spec</artifactId>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

e. 在 pom.xml 文件的 & lt;build > 项中设置以下构建配置：

```
<finalName>${project.artifactId}</finalName>
<plugins>
    <plugin>
        <groupId>org.wildfly.plugins</groupId>
        <artifactId>wildfly-jar-maven-plugin</artifactId> (1)
        <version>${bootable.jar.maven.plugin.version}</version>
        <configuration>
            <feature-pack-location>org.jboss.eap:wildfly-galleon-
pack:${jboss.xp.galleon.feature.pack.version}</feature-pack-location>
            <layers>
                <layer>jaxrs-server</layer>
                <layer>elytron-oidc-client</layer> (2)
            </layers>
            <context-root>false</context-root> (3)
```

```
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>package</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
```

1 **JBoss EAP Maven 插件，将应用程序构建为可引导 JAR**

2 **elytron-oidc-client 层提供了一个原生 OpenID Connect (OIDC)客户端来与外部 OpenID 供应商连接。**

3 **在 simple-oidc-layer-example 资源路径中注册应用程序。然后，servlet 位于 URL http://*server-url*/*application_name*/*servlet_path*，例如：http://localhost:8080/simple-oidc-layer-example/secured。默认情况下，应用程序 WAR 文件在 root-context 路径下注册，如 http://*server-url*/*servlet_path*，例如：http://localhost:8080/secured。**

f. **在 pom.xml 文件的 < build > 元素中设置应用程序名称，如 "simple-oidc-layer-example"。**

```
<finalName>simple-oidc-layer-example</finalName>
```

验证

- **在应用程序根目录中，输入以下命令：**

```
$ mvn install
```

您会看到类似如下的输出：

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

```
[INFO] Total time: 19.157 s
[INFO] Finished at: 2022-03-10T09:38:21+05:30
[INFO] ------------------------------------------------------------------------
```

现在，您可以创建一个使用 OpenID Connect 的可引导 jar 应用程序

9.4.3. 创建使用 OpenID Connect 的可引导 jar 应用程序

以下示例演示了创建一个显示已登录用户的 Principal 的 servlet。对于现有应用程序，只需要与更新部署配置相关的步骤。

在本例中，Principal 的值来自 OpenID 提供程序的 ID 令牌。默认情况下，Principal 是令牌中的 "sub" 声明的值。您可以在以下之一中指定 ID 令牌中的声明值作为 Principal ：

- elytron-oidc-client 子系统属性 principal-attribute。
- oidc.json 文件。

该流程中的 &lt ;*application_root* > 表示 pom.xml 文件目录。pom.xml 文件包含应用的 Maven 配置。

先决条件

- 您已创建了一个 Maven 项目。如需更多信息，请参阅配置 Maven 项目以创建安全应用程序。
- 您已将 Red Hat Single Sign-On 配置为 OpenID 供应商。如需更多信息，请参阅配置 Red Hat Single Sign-On 作为 OpenID 供应商。

流程

1. 创建一个用于存储 Java 文件的目录。

   语法

   -

```
$ mkdir -p <application_root>/src/main/java/com/example/oidc
```

Example

```
$ mkdir -p simple-oidc-layer-example/src/main/java/com/example/oidc
```

2. **前往新目录。**

语法

```
$ cd <application_root>/src/main/java/com/example/oidc
```

Example

```
$ cd simple-oidc-layer-example/src/main/java/com/example/oidc
```

3. **使用以下内容创建 servlet "SecuredServlet.java" :**

```
package com.example.oidc;

import java.io.IOException;
import java.io.PrintWriter;
import java.security.Principal;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * A simple secured HTTP servlet.
 *
 */
@WebServlet("/secured")
public class SecuredServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        try (PrintWriter writer = resp.getWriter()) {
            writer.println("<html>");
            writer.println("  <head><title>Secured Servlet</title></head>");
            writer.println("  <body>");
            writer.println("    <h1>Secured Servlet</h1>");
            writer.println("    <p>");
            writer.print(" Current Principal '");
            Principal user = req.getUserPrincipal();
            writer.print(user != null ? user.getName() : "NO AUTHENTICATED USER");
            writer.print("'");
            writer.println("    </p>");
            writer.println("  </body>");
            writer.println("</html>");
        }
    }

}
```

4. 在应用的 WEB-INF 目录中的部署描述符 web.xml 文件中添加访问应用程序的安全规则。

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

<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  metadata-complete="false">

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>secured</web-resource-name>
            <url-pattern>/secured</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>*</role-name>
        </auth-constraint>
    </security-constraint>

    <security-role>
```

```
    <role-name>*</role-name>
  </security-role>
</web-app>
```

5. 要使用 OpenID Connect 保护应用，可以更新部署配置或配置 elytron-oidc-client 子系统。

注意

如果您在部署配置和 elytron-oidc-client 子系统中配置 OpenID Connect，则 elytron-oidc-client 子系统 secure-deployment 属性中的配置优先于应用部署描述符中的配置。

- 更新部署配置：

  i. 在 WEB-INF 目录中创建 oidc.json 文件，如下所示：

```
{
  "provider-url" : "http://localhost:8180/auth/realms/example_realm",
  "ssl-required": "external",
  "client-id": "my_jbeap",
  "public-client": true,
  "confidential-port": 0
}
```

  ii. 使用以下文本更新部署描述符 web.xml 文件，以声明此应用程序使用 OIDC：

```
<login-config>
  <auth-method>OIDC</auth-method>
</login-config>
```

- 配置 elytron-oidc-client 子系统：

  i. 创建一个目录，将 CLI 脚本存储在应用程序根目录中：

语法

```
$ mkdir <application_root>/<cli_script_directory>
```

Example

```
$ mkdir simple-oidc-layer-example/scripts/
```

您可以在 Maven 能够访问应用程序根目录的任意位置创建该目录。

ii. 创建包含以下内容的 CLI 脚本，如 configure-oidc.cli ：

```
/subsystem=elytron-oidc-client/secure-deployment=simple-oidc-layer-
example.war:add(client-id=my_jbeap,provider-
url=http://localhost:8180/auth/realms/example_realm,public-client=true,ssl-
required=external)
```

subsystem 命令定义 simple-oidc-layer-example.war 资源，作为在 elytron-oidc-client 子系统中保护的部署。

iii. 在项目 pom.xml 文件中，将以下配置提取添加到现有插件 < configuration> 元素中：

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

6. 在应用程序根目录中，使用以下命令编译应用程序：

```
$ mvn package
```

7. 使用以下命令部署可引导 jar 应用程序：

语法

```
$ java -jar <application_root>/target/simple-oidc-layer-example-bootable.jar
```

Example

```
$ java -jar simple-oidc-layer-example/target/simple-oidc-layer-example-bootable.jar
```

这将启动 **JBoss EAP** 并部署应用程序。

验证

1. 在一个浏览器中，导航到 **http://localhost:8080/simple-oidc-layer-example/secured**。
2. 使用您的凭证登录。例如：

```
username: jane_doe
password: janedoep@$$
```

您将获得以下输出：

```
Secured Servlet
Current Principal '5cb0c4ca-0477-44c3-bdef-04db04d7e39d'
```

现在，您可以使用您在 **Red Hat Single Sign-On** 中配置的凭证作为 **OpenID** 供应商登录到应用程序。

其他资源

- JBoss EAP 中的 OpenID Connect 配置
- 根据用户角色限制对应用程序的访问

9.4.4. 根据可引导 jar OIDC 应用程序中的用户角色限制访问

您可以根据用户角色限制对应用程序的所有或部分的访问。例如，您可以让具有"public"角色的用户能够访问您应用程序的部分不敏感，并授予用户对那些部分的"admin"角色访问权限。

先决条件

- 您已使用 OpenID Connect 保护应用程序。如需更多信息，请参阅 创建使用 OpenID Connect 的可引导 jar 应用程序。

流程

1. 使用以下文本更新部署描述符 web.xml 文件：

语法

```
<security-constraint>
    ...
    <auth-constraint>
        <role-name><allowed_role></role-name>
    </auth-constraint>
</security-constraint>
```

Example

```
<security-constraint>
    ...
    <auth-constraint>
        <role-name>example_role</role-name> (1)
    </auth-constraint>
</security-constraint>
```

1

仅允许具有 **example_role** 角色的用户访问您的应用。

2. 在应用程序根目录中，使用以下命令重新编译应用程序：

```
$ mvn package
```

3. 部署应用。

```
$ java -jar simple-oidc-layer-example/target/simple-oidc-layer-example-bootable.jar
```

这将启动 **JBoss EAP** 并部署应用程序。

验证

1. 在浏览器中，导航到 **\localhost:8080/simple-oidc-layer-example/secured**。

2. 使用您的凭证登录。例如：

```
username: jane_doe
password: janedoep@$$
```

您将获得以下输出：

```
Forbidden
```

由于您尚未将所需的角色分配给用户 **"jane_doe，" jane_doe** 无法登录到您的应用程序。只有具有所需角色的用户才能登录。

要为用户分配所需的角色，请参阅 **在 Red Hat Single Sign-On 中创建和分配角色**。

9.4.5. 在 Red Hat Single Sign-On 中创建并分配用户角色

Red Hat Single Sign-On 是一个身份和访问管理供应商，用于通过单点登录(SSO)保护您的 Web 应用程序。您可以在 Red Hat Single Sign-On 中定义用户并分配角色。

先决条件

- 您已配置了 Red Hat Single Sign-On。如需更多信息，请参阅配置 Red Hat Single Sign-On 作为 OpenID 供应商。

流程

1. 登录到位于 http://localhost:<port>/auth/ 的 admin 控制台。例如：http://localhost:8180/auth/。
2. 单击用于与 JBoss EAP 连接的域。例如： *example_realm*。
3. 单击 Clients，然后单击您为 JBoss EAP 配置的 client-name。例如，*my_jbeap*。
4. 单击 Roles，然后单击 Add Role。
5. 输入角色名称，如 *example_role*，然后单击 Save。这是您在 JBoss EAP 中配置用于授权的角色名称。
6. 单击 Users，然后查看所有用户。
7. 点 ID 来分配您创建的角色。例如，单击 *jane_doe* 的 ID。
8. 点 Role Mappings。在 Client Roles 字段中，选择您为 JBoss EAP 配置的 client-name。例如，*my_jbeap*。
9. 在 Available Roles 中，选择要分配的角色。例如，*example_role*。单击 Add selected。

验证

1. 在浏览器中，导航到应用 URL。

2. 使用您的凭证登录。例如：

```
username: jane_doe
password: janedoep@$$
```

您将获得以下输出：

```
Secured Servlet
Current Principal '5cb0c4ca-0477-44c3-bdef-04db04d7e39d'
```

具有所需角色的用户可以登录到您的应用程序。

其他资源

- 使用红帽单点登录中的角色和组分配权限和访问权限

第 10 章 JBOSS EAP 中的可观察性

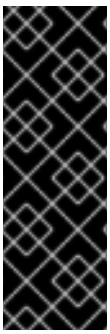
如果您是开发人员或系统管理员，则 *可观察性* 是一种一系列实践和技术，您可以根据应用中的某些信号、应用中的问题位置和来源来确定。最常见的信号是指标、事件和追踪。JBoss EAP 使用 OpenTelemetry 进行 *可观察性*。

10.1. JBOSS EAP 中的 OPENTELEMETRY

OpenTelemetry 是一组工具、应用程序编程接口(API)和软件开发套件(SDK)，可用于为应用程序检测、生成、收集和导出遥测数据。遥测数据包括指标数据、日志和追踪。分析应用程序的遥测数据可帮助您提高应用程序的性能。JBoss EAP 通过 opentelemetry 子系统提供 OpenTelemetry 功能。

注意

Red Hat JBoss Enterprise Application Platform 7.4 只提供 OpenTelemetry tracing 功能。



重要

OpenTelemetry 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。有关红帽技术预览功能支持范围的更多信息，请参阅 https://access.redhat.com/support/offerings/techpreview 。

其他资源

- OpenTelemetry 文档

10.2. JBOSS EAP 中的 OPENTELEMETRY 配置

您可以使用 opentelemetry 子系统在 JBoss EAP 中配置 OpenTelemetry 的许多方面。这包括 exporter、span 处理器和 sampler。

exporter

要分析和视觉化追踪和指标，您可以将它们导出到 Jaeger 等收集器。您可以将 JBoss EAP 配置为使用 Jaeger 或支持 OpenTelemetry 协议(OTLP)的任何收集器。

span 处理器

您可以将 span 处理器配置为导出范围，因为它们是生成或批量的。您还可以将 trace 数量配置为导出。

sampler

您可以通过配置 sampler 将 trace 数量配置为记录。

配置示例

以下 XML 是完整的 OpenTelemetry 配置的示例，包括默认值。进行更改时，JBoss EAP 不会保留默认值，因此您的配置可能会不同。

```
<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>
```

注意

您不能使用 OpenShift 路由对象来与 Jaeger 端点连接。反之，使用 http:// *<ip_address> : &* lt;*port>* 或 http:// *<service_name> : <port>*。

其他资源

- OpenTelemetry 子系统属性

10.3. JBOSS EAP 中的 OPENTELEMETRY TRACING

JBoss EAP 提供 OpenTelemetry 跟踪，可帮助您在用户请求通过应用程序的不同部分时跟踪用户请求的进度。通过分析 trace，您可以提高应用程序的性能并调试可用性问题。

OpenTelemetry 追踪由以下组件组成：

Trace

请求在应用程序中处理的操作集合。

Span

trace 中的单个操作。它提供请求、错误和持续时间(RED)指标，并包含 span 上下文。

span 上下文

代表包含 span 的一个请求的一组唯一标识符。

JBoss EAP 自动跟踪对 Jakarta RESTful Web Services 应用程序的 REST 调用，以及容器管理的 Jakarta RESTful Web 服务客户端调用。JBoss EAP 会隐式跟踪 REST 调用，如下所示：

- 对于每个传入的请求：
  - JBoss EAP 从请求中提取 span 上下文。
  - JBoss EAP 启动一个新的范围，然后在请求完成后关闭它。
- 对于每个传出请求：
  - JBoss EAP 将 span 上下文注入到请求中。
  - JBoss EAP 启动一个新的范围，然后在请求完成后关闭它。

除了隐式追踪外，您还可以通过将 Tracer 实例注入应用程序来创建自定义 span，以进行精细的追踪。

重要

如果您看到为 REST 调用导出的重复 trace，请禁用 microprofile-opentracing-smallrye 子系统。有关禁用 microprofile-opentracing-smallrye 的详情请参考 删除 microprofile-opentracing-smallrye 子系统。

其他资源

- 使用 Jaeger 观察应用程序的 OpenTelemetry 跟踪
- JBoss EAP 中的 OpenTelemetry 应用程序开发

10.4. 在 JBOSS EAP 中启用 OPENTELEMETRY 追踪

要在 JBoss EAP 中使用 OpenTelemetry tracing，您必须首先启用 opentelemetry 子系统。

先决条件

- 已安装 JBoss EAP XP。

流程

1. 使用管理 CLI 添加 OpenTelemetry 扩展。

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

2. 使用管理 CLI 启用 opentelemetry 子系统。

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

3. 重新加载 JBoss EAP。

```
reload
```

其他资源

- 配置 opentelemetry 子系统

10.5. 配置 OPENTELEMETRY 子系统

您可以配置 opentelemetry 子系统来设置追踪的不同方面。根据您用来观察追踪的收集器配置它们。

先决条件

- 您已启用了 opentelemetry 子系统。如需更多信息，请参阅 JBoss EAP 中启用 OpenTelemetry 追踪。

流程

1. 为 trace 设置导出器类型。

   语法

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

   Example

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

2. 设置要导出 trace 的端点。

   语法

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

   Example

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

3. 设置导出 trace 的服务名称。

语法

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

示例

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

其他资源

- 使用 Jaeger 观察应用程序的 OpenTelemetry 跟踪

10.6. 使用 JAEGER 观察应用程序的 OPENTELEMETRY 跟踪

JBoss EAP 自动并隐式跟踪对 Jakarta RESTful Web Services 应用程序的 REST 调用。您不需要在 Jakarta RESTful Web Services 应用中添加任何配置或配置 opentelemetry 子系统。以下流程演示了如何在 Jaeger 控制台中观察 helloworld-rs quickstart 的 trace。

先决条件

- 已安装 Docker。如需更多信息，请参阅 获取 Docker。
- 您已下载 helloworld-rs Quickstart。快速入门位于 helloworld-rs。

- 您已配置了 opentelemetry 子系统。如需更多信息，请参阅配置 opentelemetry 子系统。

流程

1. 使用其 Docker 镜像启动 Jaeger 控制台。

```
$ 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. 使用 Maven 从其根目录部署 helloworld-rs quickstart。

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

3. 在 Web 浏览器中，访问位于 http://localhost:8080/helloworld-rs/ 的快速入门，然后点任何链接。

4. 在 Web 浏览器中，打开位于 http://localhost:16686/search 的 Jaeger 控制台。hello-world.rs 列在 Service 下。

5. 选择 hello-world.rs，再单击 Find Traces。列出 hello-world.rs 的 trace 的详细信息。

其他资源

- JBoss EAP 中的 OpenTelemetry 应用程序开发

10.7. OPENTELEMETRY 追踪应用程序开发

虽然 JBoss EAP 会自动和隐式跟踪对 Jakarta RESTful Web Services 应用的 REST 调用，但您可以从应用程序中创建自定义范围以进行精细的追踪。*span* 是 trace 中的单个操作。例如，您可以在应用程

序中创建一个范围，例如定义资源，称为方法，以此类推。您可以通过注入 Tracer 实例来在应用程序中创建自定义 trace。

10.7.1. 为 OpenTelemetry tracing 配置 Maven 项目

要创建 OpenTelemetry 追踪应用程序，请创建一个具有所需依赖项和目录结构的 Maven 项目。

先决条件

- 您已安装了 Maven。如需更多信息，请参阅 下载 Apache Maven。
- 您已为最新版本配置了 Maven 存储库。有关安装最新的 Maven 存储库补丁的详情，请参考 Maven 和 JBoss EAP microprofile maven 存储库。

流程

1. 在 CLI 中，使用 mvn 命令来设置 Maven 项目。此命令为项目创建目录结构，以及 pom.xml 配置文件。

   语法

```
$ 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. 导航到应用程序根目录。

语法

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

Example

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

3. 更新生成的 pom.xml 文件。

a. 设置以下属性：

```
<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. 设置以下依赖项：

```
<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. **将以下构建配置设置为使用 mvn widlfy:deploy 来部署应用程序：**

```
<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>
```

验证

- **在应用程序根目录中，输入以下命令：**

```
$ mvn install
```

您会看到类似如下的输出：

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.440 s
[INFO] Finished at: 2021-12-27T14:45:12+05:30
[INFO] ------------------------------------------------------------------------
```

现在，您可以创建一个 OpenTelemetry 追踪应用程序。

其他资源

- 创建创建自定义范围的应用程序

10.7.2. 创建创建自定义范围的应用程序

以下流程演示了如何创建可创建两个自定义 span 的应用程序，如下所示：

- prepare-hello - 当应用中的方法 getHello （） 被调用时。
- process-hello - 当值 hello 分配给一个新的 String 对象 hello 时。

此流程还演示了如何在 Jaeger 控制台中查看这些 span。该流程中的 & lt;application_root > 表示包含 pom.xml 文件的目录，其中包含应用程序的 Maven 配置。

先决条件

- 已安装 Docker。如需更多信息，请参阅 获取 Docker。
- 您已创建了一个 Maven 项目。如需更多信息，请参阅为 OpenTelemetry tracing 配置 Maven 项目。
- 您已配置了 opentelemetry 子系统。如需更多信息，请参阅配置 opentelemetry 子系统。

流程

1. 在 & *lt;application_root* > 中，创建一个用于存储 Java 文件的目录。

   语法

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

-

Example

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

2. **前往新目录。**

语法

```
$ cd src/main/java/com/example/opentelemetry
```

Example

```
$ cd src/main/java/com/example/opentelemetry
```

3. **创建包含以下内容的 JakartaRestApplication.java 文件：此 JakartaRestApplication 类将应用声明为 Jakarta RESTful Web Services 应用。**

```
package com.example.opentelemetry;

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

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

4. 创建包含以下内容的 ExplicitlyTracedBean.java 文件，其内容为 class ExplicitlyTracedBean。此类通过注入 Tracer 类来创建自定义 span。

```
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;
   }
}
```

(1) 注入 Tracer 类以创建自定义范围。

(2) 创建名为 prepare-hello 的 span，以指示方法 getHello （） 被调用。

(3) 创建名为 process-hello 的 span，以指示值 hello 已分配给名为 hello 的新 String 对象。

5. 为 TracedResource.java 类创建包含以下内容的 TracedResource.java 文件。此文件注入 ExplicitlyTracedBean 类，并声明两个端点： traced 和 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. 导航到应用程序根目录。

语法

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

Example

```
$ cd ~/applications/simple-tracing-example
```

7. **使用以下命令编译并部署应用程序：**

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

8. **启动 Jaeger 控制台。**

```
$ 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. **在浏览器中，导航到 \localhost:8080/simple-tracing-example/hello/cdi-trace。**

10. **在一个浏览器中，打开位于 http://localhost:16686/search 的 Jaeger 控制台。**

11. **在 Jaeger 控制台中，选择 JBoss EAP XP 并点 Find Traces。**

12. **单击 3 Spans。**

13. **Jaeger 控制台显示以下 trace：**

```
|GET /hello/cdi-trace 1
-
 | prepare-hello 2
  -
   | process-hello 3
```

1

这是自动隐式追踪的范围。

2

自定义 span prepare-hello 表示方法 getHello （） 被调用。它是自动隐式追踪的 span 子级。

3

自定义 span process-hello 表示值 hello 已分配给一个新的 String 对象 hello。它是 prepare-hello 范围的子项。

每当您访问 http://localhost:16686/search 的应用程序端点时，都会使用所有子范围创建一个新的 trace。

其他资源

- JBoss EAP 中的 OpenTelemetry tracing

第 11 章 参考

11.1. MICROPROFILE CONFIG 参考

11.1.1. 默认 MicroProfile Config 属性

MicroProfile Config 规范默认定义三个 ConfigSources。

ConfigSources 根据常规数字进行排序。如果需要为后续部署覆盖配置，则在较高情况的 ConfigSource 之前会覆盖较低或dinal ConfigSource。

表 11.1. 默认 MicroProfile Config 属性

ConfigSource	ordinal
系统属性	400
环境变量	300
classpath 上找到的属性文件 META-INF/microprofile-config.properties	100

11.1.2. MicroProfile Config SmallRye ConfigSources

microprofile-config-smallrye 项目定义了除默认的 MicroProfile ConfigSources 外，您还可以使用的更多 ConfigSources。

表 11.2. 其他 MicroProfile 配置属性

ConfigSource	ordinal
subsystem 中的 config-source	100
目录中的 ConfigSource	100
来自类的 ConfigSource	100

没有为这些 ConfigSources 指定明确的 ordinal。它们继承 MicroProfile 配置规范中找到的默认值。

11.2. MICROPROFILE FAULT TOLERANCE 参考

11.2.1. MicroProfile Fault Tolerance 配置属性

小型容错规范除了 MicroProfile 容错规范中定义的属性外，还定义了下列属性：

表 11.3. MicroProfile Fault Tolerance 配置属性

属性	默认值	描述
io.smallrye.faulttolerance.mainThreadPoolSize	**100**	线程池中的最大线程数量。
io.smallrye.faulttolerance.mainThreadPoolQueueSize	**-1 (unbounded)**	线程池应使用的队列的大小。

11.3. MICROPROFILE JWT 参考

11.3.1. MicroProfile Config JWT 标准属性

microprofile-jwt-smallrye 子系统支持以下 MicroProfile 配置标准属性：

表 11.4. MicroProfile Config JWT 标准属性

属性	默认	描述
mp.jwt.verify.publickey	NONE	使用其中一个支持的格式编码的公钥的字符串。如果您设置了 **mp.jwt.verify.publickey.location**，则不要设置。
mp.jwt.verify.publickey.location	NONE	公钥的位置可以是相对路径或 URL。如果您设置了 **mp.jwt.verify.publickey**，则不要设置。
mp.jwt.verify.issuer	NONE	任何 JWT 令牌的预期值 **是被** 验证的 JWT 令牌的声明。

microprofile-config.properties 配置示例：

```
mp.jwt.verify.publickey.location=META-INF/public.pem
mp.jwt.verify.issuer=jwt-issuer
```

11.4. MICROPROFILE OPENAPI 参考

11.4.1. MicroProfile OpenAPI 配置属性

除了标准 MicroProfile OpenAPI 配置属性外，JBoss EAP 还支持下列额外 MicroProfile OpenAPI 属性：这些属性可以在全局和应用范围内应用。

表 11.5. JBoss EAP 中的 MicroProfile OpenAPI 属性

属性	默认值	描述
mp.openapi.extensions.enabled	**true**	启用或禁用 OpenAPI 端点的注册。 当设置为 **false** 时，禁用 OpenAPI 文档的生成。您可以使用 config 子系统或配置文件中每个应用（如 **/META-INF/microprofile-config.properties** ）设置值。 您可以对此属性进行参数化，以选择性地在不同环境中启用或禁用 **microprofile-openapi-smallrye**，如生产或开发。 您可以使用此属性来控制与给定虚拟主机关联的应用应生成 MicroProfile OpenAPI 模型。
mp.openapi.extensions.path	**/openapi**	您可以使用此属性为与虚拟主机关联的多个应用程序生成 OpenAPI 文档。 在与同一虚拟主机关联的每个应用程序上设置不同的 **mp.openapi.extensions.path** 。
mp.openapi.extensions.servers.relative	**true**	指明自动生成的服务器记录是绝对或相对于 OpenAPI 端点的位置。 需要服务器记录来确保存在非根上下文路径，OpenAPI 文档的用户可以构建与 OpenAPI 端点主机相关的 REST 服务的有效 URL。 **true** 表示服务器记录相对于 OpenAPI 端点的位置。生成的记录包含部署的上下文路径。 当设置为 **false** 时，JBoss EAP XP 生成服务器记录，包括可以访问部署的所有协议、主机和端口。

11.5. MICROPROFILE REACTIVE MESSAGING 参考

11.5.1. MicroProfile 被动消息传递连接器，用于与外部消息传递系统集成

以下是 MicroProfile 配置规范所需的被动消息传递属性密钥前缀的列表：

- mp.messaging.incoming.[channel-name].[attribute]=[value]
- mp.messaging.outgoing.[channel-name].[attribute]=[value]
- mp.messaging.connector.[connector-name].[attribute]=[value]

请注意，channel-name 是 @Incoming.value （） 或 @Outgoing.value （）。要清楚起见，请查看这个一组连接器方法示例：

```
@Outgoing("to")
public int send() {
  int i = // Randomly generated...
  return i;
}

@Incoming("from")
public void receive(int i) {
  // Process payload
}
```

在本例中，所需的属性前缀如下：

- mp.messaging.incoming.from.这定义了 receive （） 方法。
- mp.messaging.outgoing.to.这将定义 send （） 方法。

请记住，这是一个示例。因为不同的连接器可识别不同的属性，所以您指示的前缀取决于您要配置的连接器。

11.5.2. 重新主动消息传递流和用户重新初始化代码之间的数据交换示例

以下是重新主动消息传递流和通过 @Channel 和 Emitter 结构触发的代码之间的数据交换示例：

```
@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

    }
}
```

行中详情：

1

嵌套构造器的发布程序。

2

您需要此空构造器来满足 Java 规范的上下文和依赖注入(CDI)。

3

订阅委派。

4

在可以处理多个订阅的发布程序中嵌套委托。

5

嵌套发布程序从委派中转发数据。

6

取消订阅被动消息传递提供的发布程序。

在本例中，MicroProfile Reactive Messaging 正在侦听 my-stream 内存流，因此通过 Emitter 发送的消息会收到注入的发布者。但请注意，需要满足以下条件才能使这个数据交换成功：

1. 在调用 Emitter.send （） 之前，该频道必须具有有效的订阅。在本例中，请注意构造器调用的 subscribeAndAllowMultipleSubscriptions （） 方法确保 bean 可用于用户代码调用时具有活跃的订阅。

2. 在注入的 publisher 中只能有一个 订阅 。如果要使用 REST 调用公开接收发布程序，其中每个调用 poll （） 方法会导致对 dest publisher 的新订阅，您必须实施自己的发布程序，以便从注入的每个客户端广播数据。

11.5.3. Apache Kafka 用户 API

您可以使用 Apache Kafka 用户 API 来获取有关收到消息 Kafka 的更多信息，并影响 Kafka 处理消息的方式。此 API 存储在 io/smallrye/reactive/messaging/kafka/api 软件包中，它由以下类组成：

- IncomingKafkaRecordMetadata.这个元数据包含以下信息：
  - Kafka 记录 密钥，由 消息 表示。
  - 用于 消息的 Kafka 主题和 分区 ， 以及这些 消息中 的偏移量。
  - Message timestamp 和 timestampType。

- Message 标头.这些是应用程序可在生成端附加的信息，并在消费端接收信息。

- OutgoingKafkaRecordMetadata.使用这个元数据，您可以指定或覆盖 Kafka 处理信息的方式。它包含以下信息：

  - Kafka 视为消息键的密钥。

  - 您希望 Kafka 使用 的主题。

  - 分区。

  - 时间戳，如果您不希望 Kafka 生成 的时间戳。

  - 标头.

- KafkaMetadataUtil 包含将 OutgoingKafkaRecordMetadata 写入 Message 的工具方法，并从 Message 读取 IncomingKafkaRecordMetadata。

重要

如果您将 OutgoingKafkaRecordMetadata 写入发送到未映射到 Kafka 的频道的消息，则被动消息传递框架会忽略它。相反，如果您从未映射到 Kafka 的频道 读取 IncomingKafkaRecordMetadata，则该消息会返回 null。

如何写入和读取消息 键的示例

```
@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();
}
```

microprofile-config.properties 文件中的 Kafka 映射示例

```
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.StringSerializer

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.serialization.IntegerDeserializer
mp.messaging.incoming.from-
kafka.key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
```

注意

您必须为传出频道指定 key.serializer，并为传入频道指定 key.deserializer。

11.5.4. Kafka 连接器的 MicroProfile Config 属性文件示例

这是 **Kafka 连接器的简单 microprofile-config.properties 文件示例。其属性与示例中的属性对应 "MicroProfile 重新主动消息传递连接器，以与外部消息传递系统集成"。**

```
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.common.serialization.IntegerSerializer
```

```
mp.messaging.incoming.from.connector=smallrye-kafka
mp.messaging.incoming.from.topic=my-topic
mp.messaging.incoming.from.value.deserializer=org.apache.kafka.common.serialization.IntegerDeserializer
```

表 11.6. 条目的讨论

entry	描述
到，从	这些是"频道"。
发送,receive	这些是"方法"。 请注意，**要** 频道位于 **send （）** 方法，而 **from** 频道则位于 **receive （）** 方法。
kafka.bootstrap.servers=kafka:9092	这将指定应用程序必须连接到的 Kafka 代理的 URL。您还可以在频道级别指定一个 URL，如下所示：**mp.messaging.outgoing.to.bootstrap.servers=kafka:9092**
mp.messaging.outgoing.to.connector=smallrye-kafka	这表示您希望要频道从 Kafka 接收信息。 **小型被动消息传递是用于构建应用程序的框架。请注意，smallrye-kafka 值是 SmallRye 特定于重新主动的消息传递。如果您使用 Galleon 置备自己的服务器，您可以通过包含 microprofile-reactive-messaging-kafka Galleon 层来启用 Kafka 集成。**
mp.messaging.outgoing.to.topic=my-topic	这表示您要将数据发送到名为 **my-topic** 的 Kafka 主题。 Kafka "topic" 是一个类别或源名称，信息存储在并发布到其中。所有 Kafka 信息都组织为主题。生产者应用将数据写入主题和消费者应用，*从*主题中读取数据。
mp.messaging.outgoing.to.value.serializer=org.apache.kafka.common.serialization.IntegerSerializer	这告知连接器使用 **IntegerSerializer** 来序列化 **send （）** 方法输出的值。Kafka 为标准 Java 类型提供序列化器。您可以通过编写实现 **org.apache.kafka.common.serialization.Serializer** 的类来实现自己的序列化器，然后在部署中包含该类。
mp.messaging.incoming.from.connector=smallrye-kafka	这表示您要使用 **from** 频道来接收来自 Kafka 的信息。同样，**smallrye-kafka** 的值是 SmallRye reactive messaging-specific。

entry	描述
mp.messaging.incoming.from.topic=my-topic	这表示您的连接器应该从名为 **my-topic** 的 Kafka 主题中读取数据。
mp.messaging.incoming.from.value.deserializer=org.apache.kafka.common.serialization.IntegerDeserializer	这告知连接器在调用 **receive （）** 方法前，使用 **IntegerDeserializer** 来反序列化主题中的值。您可以通过编写实现 **org.apache.kafka.common.serialization.Deserializer** 的类来实现自己的反序列化器，然后在部署中包含该类。

注意

这个属性列表并不全面。如需更多信息，请参阅 SmallRye Reactive Messaging Apache Kafka 文档。

强制 MicroProfile 主动消息传递前缀

MicroProfile Reactive 消息传递规范需要 Kafka 的以下方法属性密钥前缀：

- **mp.messaging.incoming.[channel-name].[attribute]=[value]`**
- **mp.messaging.outgoing.[channel-name].[attribute]=[value]`**
- **mp.messaging.connector.[connector-name].[attribute]=[value]`**

请注意，channel-name 是 @Incoming.value （） 或 @Outgoing.value （）。

现在考虑以下方法对示例：

```
@Outgoing("to")
public int send() {
    int i = // Randomly generated...
    return i;
}

@Incoming("from")
public void receive(int i) {
    // Process payload
}
```

在这个方法对示例中，请注意以下所需属性前缀：

- mp.messaging.incoming.from.此前缀选择属性作为 receive （） 方法的配置。
- mp.messaging.outgoing.to.此前缀选择属性作为 send （） 方法的配置。

11.6. OPENID CONNECT 参考

11.6.1. Elytron-oidc-client 子系统属性

elytron-oidc-client 子系统提供属性来配置其行为。

表 11.7. Elytron-oidc-client 子系统属性

属性	描述
provider	配置 OpenID Connect 供应商。
secure-deployment	由 OpenID Connect 供应商保护的部署。
realm	配置 Red Hat Single Sign-On 域。这为方便用户提供。您可以在 keycloak 客户端适配器中复制配置，并在此处使用。建议使用 **供应商**。

重要

不要在您的配置中使用以下 供应商、realm 和 secure-deployment 属性，因为目前不支持它们：

- **autodetect-bearer-only**
- **bearer-only**

不要在配置中使用以下 secure-deployment 属性，因为目前不支持它

- **enable-basic-auth**

为以下目的使用三个 elytron-oidc-client 属性：

- **Provider: 用于配置 OpenID Connect 供应商。如需更多信息，请参阅 供应商 属性。**
- **secure-deployment ：用于配置由 OpenID Connect 保护的部署。如需更多信息，请参阅 secure-deployment 属性**
- **域 ：用于配置 Red Hat Single Sign-On。如需更多信息，请参阅 域 属性。不建议使用 realm。它是为了方便使用。您可以在 keycloak 客户端适配器中复制配置，并在此处使用。建议使用 provider 属性。**

表 11.8. 供应商 属性

属性	默认值	描述
allow-any-hostname	**false**	如果将值设为 **true**，则在与 OpenID 供应商通信时跳过主机名验证。这在测试时很有用。不要将其设置为生产环境中的 **ture**。
always-refresh-token		如果设置为 **true**，JBoss EAP 会在每次 Web 请求上刷新令牌。

属性	默认值	描述
auth-server-url		Red Hat Single Sign-On 域授权服务器的基本 URL。如果使用此属性，还必须定义 **realm** 属性。 您还可以使用 **provider-url** 属性在单个属性中提供基本 URL 和 realm。
client-id		在 OpenID 提供程序中注册的 JBoss EAP 的客户端 ID。
client-key-password		如果指定了 **client-keystore**，请在此属性中指定密码。
client-keystore		如果您的应用通过 HTTPS 与 OpenID 提供程序通信，请在此属性中设置客户端密钥存储的路径。
client-keystore-password		如果指定了 **客户端密钥存储**，请提供用于在此属性中访问它的密码。
confidential-port	**8443**	指定 OpenID 供应商使用的机密端口(SSL/TLS)。
connection-pool-size		指定与 OpenID 供应商通信时使用的连接池大小。
connection-timeout-millis		指定与远程主机建立连接的超时时间（以毫秒为单位）。最小值为 **-1L**，最大 **2147483647L**.**-1L** 表示该值未定义，这是默认值。
connection-ttl-millis		指定连接保留的时间（以毫秒为单位）。最小值为 **-1L**，最大 **2147483647L**。**-1L** 表示值未定义，这是默认值。
cors-allowed-headers		如果启用了 Cross-Origin Resource Sharing (CORS)，这将设置 **Access-Control-Allow-Headers** 标头的值。这应该是一个用逗号分开的字符串。这是可选的。如果没有设置，则不会在 CORS 响应中返回此标头。
cors-allowed-methods		如果启用了 Cross-Origin Resource Sharing (CORS)，这将设置 Access-Control-Allow-Methods 标头的值。这应该是一个用逗号分开的字符串。这是可选的。如果没有设置，则不会在 CORS 响应中返回此标头。
cors-exposed-headers		如果启用了 CORS，这将设置 Access-Control-Expose-Headers 标头的值。这应该是一个用逗号分开的字符串。这是 optinal。如果没有设置，则不会在 CORS 响应中返回此标头。
cors-max-age		设置 Cross-Origin Resource Sharing (CORS) Max-Age 标头的值。该值可以在 **-1L** 和 **2147483647L** 之间。只有在 **enable-cors** 设为 **true** 时，此属性才会生效。
disable-trust-manager		指定在通过 HTTPS 与 OpenID 供应商通信时是否使用信任管理器。

属性	默认值	描述
enable-cors	**false**	启用 Red Hat Single Sign-On Cross-Origin Resource Sharing (CORS)支持。
expose-token	**false**	如果设置为 **true**，经过身份验证的浏览器客户端可以通过 Javascript HTTP 调用来获取签名的访问令牌，通过 URL **root/k_query_bearer_token**。这是可选的。这特定于 Red Hat Single Sign-On。
ignore-oauth-query-parameter	**false**	禁用 access_token 的查询参数解析。
principal-attribute		指定 ID 令牌中的声明值，用作身份的主体
provider-url		指定 OpenID 供应商 URL。
proxy-url		如果使用 HTTP 代理，请指定 HTTP 代理的 URL。
realm-public-key		指定域的公钥。
register-node-at-startup	**false**	如果设置为 **true**，则会将注册请求发送到 Red Hat Single Sign-On。此属性仅在您的应用程序集群时才有用。
register-node-period		指定重新注册节点的频率。
socket-timeout-millis		以毫秒为单位指定等待数据的套接字超时。
ssl-required	**external**	指定与 OpenID 供应商的通信是否应该通过 HTTPS。该值可以是以下之一： • **All** - 所有通信都通过 HTTPS 进行。 • **external** - 只有与外部客户端的通信通过 HTTP 进行。 • **none** - 未使用 HTTP。
token-signature-algorithm	**RS256**	指定 OpenID 供应商使用的令牌签名算法。支持的算法有： • **RS256** • **RS384** • **RS512** • **ES256** • **ES384** • **ES512**
token-store		为 auth-session 数据指定 Cookie 或会话存储。

属性	默认值	描述
truststore		指定用于客户端 HTTPS 请求的信任存储。
truststore-password		指定 truststore 密码。
verify-token-audience	**false**	如果设置为 **true**，则在仅 bearer 身份验证期间，如果令牌包含此客户端名称(**资源**)作为受众，则验证。

表 11.9. secure-deployment 属性

属性	默认值	描述
allow-any-hostname	**false**	如果将值设为 **true**，则在与 OpenID 供应商通信时跳过主机名验证。这在测试时很有用。不要将其设置为生产环境中的 **ture**。
always-refresh-token		如果设置为 **true**，JBoss EAP 会在每次 Web 请求上刷新令牌。
auth-server-url		Red Hat Single Sign-On 域授权服务器的基本 URL 可以使用 **provider-url** 属性。
client-id		在 OpenID 提供程序中注册的 JBoss EAP 的客户端 ID。
client-key-password		如果指定了 **client-keystore**，请在此属性中指定密码。
client-keystore		如果您的应用通过 HTTPS 与 OpenID 提供程序通信，请在此属性中设置客户端密钥存储的路径。
client-keystore-password		如果指定了 **客户端密钥存储**，请提供用于在此属性中访问它的密码。
confidential-port	**8443**	指定 OpenID 供应商使用的机密端口(SSL/TLS)。
connection-pool-size		指定与 OpenID 供应商通信时使用的连接池大小。
connection-timeout-millis		指定与远程主机建立连接的超时时间（以毫秒为单位）。最小值为 **-1L**，最大 **2147483647L**。**-1L** 表示值未定义，这是默认值。

属性	默认值	描述
connection-ttl-millis		指定连接保留的时间（以毫秒为单位）。最小值为 **-1L**，最大 **2147483647L**. -1L 表示该值未定义，这是默认值。
cors-allowed-headers		如果启用了 Cross-Origin Resource Sharing (CORS)，这将设置 **Access-Control-Allow-Headers** 标头的值。这应该是一个用逗号分开的字符串。这是可选的。如果没有设置，则不会在 CORS 响应中返回此标头。
cors-allowed-methods		如果启用了 Cross-Origin Resource Sharing (CORS)，这将设置 **Access-Control-Allow-Methods** 标头的值。这应该是一个用逗号分开的字符串。这是可选的。如果没有设置，则不会在 CORS 响应中返回此标头。
cors-exposed-headers		如果启用了 Cross-Origin Resource Sharing (CORS)，这将设置 **Access-Control-Expose-Headers** 标头的值。这应该是一个用逗号分开的字符串。这是可选的。如果没有设置，则不会在 CORS 响应中返回此标头。
cors-max-age		设置 Cross-Origin Resource Sharing (CORS) Max-Age 标头的值。该值可以在 **-1L** 和 **2147483647L** 之间。此属性仅在 'enable-- 时才生效。
credential		指定用于与 OpenID 供应商通信的凭证。
disable-trust-manager		指定在通过 HTTPS 与 OpenID 供应商通信时是否使用信任管理器。
enable-cors	**false**	启用 Red Hat Single Sign-On Cross-Origin Resource Sharing (CORS)支持。

属性	默认值	描述
expose-token	**false**	如果设置为 **true**，经过身份验证的浏览器客户端可以通过 Javascript HTTP 调用来获取签名的访问令牌，通过 URL **root/k_query_bearer_token**。这是可选的。这特定于 Red Hat Single Sign-On。
ignore-oauth-query-parameter	**false**	禁用 access_token 的查询参数解析。
min-time-between-jwks-requests		如果适配器识别了由未知公钥签名的令牌，JBoss EAP 会尝试从 **elytron-oidc-client** 服务器下载新公钥。但是，如果您已经尝试了小于这个值的值，则 JBoss EAP deosn 不会尝试下载新的公钥，以秒为单位。该值可以在 **-1L** 和 **2147483647L** 之间。
principal-attribute		指定 ID 令牌中的声明值，用作身份的主体
provider		指定 OpenID 供应商。
provider-url		指定 OpenID 供应商 URL。
proxy-url		如果使用 HTTP 代理，请指定 HTTP 代理的 URL。
public-client	**false**	如果设置为 **true**，则在与 OpenID 提供程序通信时不会发送客户端凭证。这是可选的。
realm		在 Red Hat Single Sign-On 中连接的域。
realm-public-key		指定域的公钥。
redirect-rewrite-rule		指定要应用到重定向 URI 的重写规则。
register-node-at-startup	**false**	如果设置为 **true**，则会将注册请求发送到 Red Hat Single Sign-On。此属性仅在您的应用程序集群时才有用。

属性	默认值	描述
register-node-period		指定重新注册节点的频率。
resource		指定您要使用 OIDC 保护的应用程序名称。或者，您可以指定 **client-id**。
socket-timeout-millis		以毫秒为单位指定等待数据的套接字超时。
ssl-required	**external**	指定与 OpenID 供应商的通信是否应该通过 HTTPS。该值可以是以下之一： • **All** - 所有通信都通过 HTTPS 进行。 • **external** - 只有与外部客户端的通信通过 HTTP 进行。 • **none** - 未使用 HTTP。
token-minimum-time-to-live		如果当前令牌过期或是在您设定的时间（以秒为单位）内过期，则适配器会刷新令牌。
token-signature-algorithm	**RS256**	指定 OpenID 供应商使用的令牌签名算法。支持的算法有： • **RS256** • **RS384** • **RS512** • **ES256** • **ES384** • **ES512**
token-store		为 auth-session 数据指定 Cookie 或会话存储。
truststore		指定用于适配器客户端 HTTPS 请求的信任存储。
truststore-password		指定 truststore 密码。
turn-off-change-session-id-on-login	**false**	会话 ID 默认在成功登录时更改。将值设为 **true** 以将此关闭。

属性	默认值	描述
use-resource-role-mappings	**false**	使用从令牌获取的资源级别权限。
verify-token-audience	**false**	如果设置为 **true**，则在仅 bearer 身份验证过程中，适配器会验证令牌是否包含这个客户端名称(**资源**)作为受众。

表 11.10. 域 属性

属性	默认值	描述
allow-any-hostname	**false**	如果将值设为 **true**，则在与 OpenID 供应商通信时跳过主机名验证。这在测试时很有用。不要将其设置为生产环境中的 **ture**。
always-refresh-token		如果设置为 **true**，JBoss EAP 会在每次 Web 请求上刷新令牌。
auth-server-url		Red Hat Single Sign-On 域授权服务器的基本 URL 可以使用 **provider-url** 属性。
client-key-password		如果指定了 **client-keystore**，请在此属性中指定密码。
client-keystore		如果您的应用通过 HTTPS 与 OpenID 提供程序通信，请在此属性中设置客户端密钥存储的路径。
client-keystore-password		如果指定了 **客户端密钥存储**，请提供用于在此属性中访问它的密码。
confidential-port	**8443**	指定红帽单点登录使用的机密端口 (SSL/TLS)。
connection-pool-size		指定与 Red Hat Single Sign-On 通信时使用的连接池大小。
connection-timeout-millis		指定与远程主机建立连接的超时时间（以毫秒为单位）。最小值为 **-1L**，最大 **2147483647L**。**-1L** 表示值未定义，这是默认值。

属性	默认值	描述
connection-ttl-millis		指定连接保留的时间（以毫秒为单位）。最小值为 **-1L**，最大 **2147483647L**。**-1L** 表示值未定义，这是默认值。
cors-allowed-headers		如果启用了 Cross-Origin Resource Sharing (CORS)，这将设置 Access-Control-Allow-Headers 标头的值。这应该是一个用逗号分开的字符串。这是可选的。如果没有设置，则不会在 CORS 响应中返回此标头。
cors-allowed-methods		如果启用了 Cross-Origin Resource Sharing (CORS)，这将设置 **Access-Control-Allow-Methods 标头的值**。这应该是一个用逗号分开的字符串。这是可选的。如果没有设置，则不会在 CORS 响应中返回此标头。
cors-exposed-headers		如果启用了 Cross-Origin Resource Sharing (CORS)，这将设置 **Access-Control-Expose-Headers** 标头的值。这应该是一个用逗号分开的字符串。这是可选的。如果没有设置，则不会在 CORS 响应中返回此标头。
cors-max-age		设置 Cross-Origin Resource Sharing (CORS) Max-Age 标头的值。该值可以在 **-1L** 和 **2147483647L** 之间。只有在 **enable-cors** 设为 **true** 时，此属性才会生效。
disable-trust-manager		指定在通过 HTTPS 与 OpenID 供应商通信时是否使用信任管理器。
enable-cors	**false**	启用 {RHProductShortName} Cross-Origin Resource Sharing (CORS)支持。

属性	默认值	描述
expose-token	**false**	如果设置为 **true**，经过身份验证的浏览器客户端可以通过 Javascript HTTP 调用来获取签名的访问令牌，通过 URL **root/k_query_bearer_token**。这是可选的。
ignore-oauth-query-parameter	**false**	禁用 access_token 的查询参数解析。
principal-attribute		指定 ID 令牌中的声明值，用作身份的主体
provider-url		指定 OpenID 供应商 URL。
proxy-url		如果使用 HTTP 代理，请指定 HTTP 代理的 URL。
realm-public-key		指定域的公钥。
register-node-at-startup	**false**	如果设置为 **true**，则会将注册请求发送到 Red Hat Single Sign-On。此属性仅在您的应用程序集群时才有用。
register-node-period		指定重新注册节点的频率。
socket-timeout-millis		以毫秒为单位指定等待数据的套接字超时。
ssl-required	**external**	指定与 OpenID 供应商的通信是否应该通过 HTTPS。该值可以是以下之一： • **All** - 所有通信都通过 HTTPS 进行。 • **external** - 只有与外部客户端的通信通过 HTTP 进行。 • **none** - 未使用 HTTP。

属性	默认值	描述
token-signature-algorithm	**RS256**	指定 OpenID 供应商使用的令牌签名算法。支持的算法有： • **RS256** • **RS384** • **RS512** • **ES256** • **ES384** • **ES512**
token-store		为 auth-session 数据指定 Cookie 或会话存储。
truststore		指定用于客户端 HTTPS 请求的信任存储。
truststore-password		指定 truststore 密码。
verify-token-audience	**false**	如果设置为 **true**，则在仅 bearer 身份验证过程中，适配器会验证令牌是否包含这个客户端名称（资源）作为 audience。

其他资源

- **JBoss EAP 中的 OpenID Connect 配置**
- **使用 OpenID Connect 与红帽单点登录保护应用程序**

11.7. OPENTELEMETRY 参考

11.7.1. OpenTelemetry 子系统属性

您可以修改 opentelemetry 子系统属性来配置其行为。属性按它们配置的不同方面分组： exporter、sampler 和 span 处理器。

表 11.11. 导出器属性组

属性	描述	默认值
端点	OpenTelemetry 推送 trace 的 URL。把它设置为您的 exporter 侦听的 URL。	http://localhost:14250/
exporter-type	将 trace 发送到的导出器。它可以是以下之一： • **Jaeger**.您使用的 exporter 是 Jaeger。 • **OTLP**.您使用的导出器可用于 OpenTelemetry 协议。	**jaeger**

表 11.12. sampler 属性组

属性	描述	默认值
比率	要导出的 trace 的比率。该值必须在 **0.0** 和 **1.0** 之间。例如，要在由应用程序创建的每 100 个 trace 中导出一个 trace，请将值设为 **0.01**。只有在将属性 **sampler-type** 设置为 **ratio** 时，此属性才会生效。	

表 11.13. span 处理器属性组

属性	描述	默认值
batch-delay	JBoss EAP 连续两个导出间隔（毫秒）。只有在将属性 **span-processor-type** 设置为 **batch** 时，此属性才会生效。	**5000**
export-timeout	在取消前允许导出完成的最大时间（毫秒）。	**30000**
max-export-batch-size	每个批处理中发布的 trace 的最大数量。这个数字应该较低或等于 **max-queue-size** 的值。只有在将属性 **span-processor-type** 设置为 **batch** 时，才能设置此属性。	**512**

属性	描述	默认值
max-queue-size	导出前到队列的最大 trace 数量。如果应用程序创建更多 trace，则不会记录它们。只有在将属性 **span-processor-type** 设置为 **batch** 时，此属性才会生效。	**2048**

span-processor-type	要使用的范围处理器的类型。该值可以是以下之一：
	• **批处理** ：JBoss EAP 在使用以下属性定义的批处理中导出 trace：
	  ◦ **batch-delay**
	  ◦ **max-export-batch-size**
	  ◦ **max-queue-size**
	• **简单** ：JBoss EAP 在完成后就立即导出 trace。
		batch

其他资源

- JBoss EAP 中的 OpenTelemetry