Red Hat JBoss Enterprise Application Platform 7.4

使用 JBoss EAP XP 3.0.0

用于 JBoss EAP XP 3.0.0

Last Updated: 2023-08-28

Red Hat JBoss Enterprise Application Platform 7.4 使用 JBoss EAP XP 3.0.0

用于 JBoss EAP XP 3.0.0

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律通告

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Using_JBoss_EAP_XP_3.0.0.ent file |.

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

摘要

本文档提供有关在 JBoss EAP XP 3.0.0 中使用 MicroProfile 的常规信息。

目录

使开源包含更多 6

对红帽文档提供反馈 7

第 1 章 适用于最新 MICROPROFILE 功能的 JBOSS EAP XP 8
- 1.1. 关于 JBOSS EAP XP 8
- 1.2. JBOSS EAP XP 安装 8
- 1.3. JBOSS EAP XP 管理器 9
- 1.4. JBOSS EAP XP MANAGER 3.0 命令 9
- 1.5. 在 JBOSS EAP 7.4.X 上安装 JBOSS EAP XP 3.0.0 11
- 1.6. 卸载 JBOSS EAP XP 12
- 1.7. 查看 JBOSS EAP XP 的状态 13
- 1.8. 回滚 JBOSS EAP XP 和 JBOSS EAP 7.4.X 基本补丁 13

第 2 章 了解 MICROPROFILE 14
- 2.1. MICROPROFILE CONFIG 14
  - 2.1.1. JBoss EAP 中的 MicroProfile 配置 14
  - 2.1.2. MicroProfile 配置中支持的 MicroProfile 配置源 14
- 2.2. MICROPROFILE 容错 15
  - 2.2.1. 关于 MicroProfile 容错规范 15
  - 2.2.2. JBoss EAP 中的 MicroProfile 容错 15
- 2.3. MICROPROFILE HEALTH 16
  - 2.3.1. JBoss EAP 中的 MicroProfile 健康 16
- 2.4. MICROPROFILE JWT 17
  - 2.4.1. JBoss EAP 中的 MicroProfile JWT 集成 17
  - 2.4.2. 传统部署和 MicroProfile JWT 部署之间的区别 17
  - 2.4.3. JBoss EAP 中的 MicroProfile JWT 激活 17
  - 2.4.4. JBoss EAP 中 MicroProfile JWT 的限制 18
- 2.5. MICROPROFILE METRICS 18
  - 2.5.1. JBoss EAP 中的 MicroProfile 指标 18
- 2.6. MICROPROFILE OPENAPI 18
  - 2.6.1. JBoss EAP 中的 MicroProfile OpenAPI 18
- 2.7. MICROPROFILE OPENTRACING 19
  - 2.7.1. MicroProfile OpenTracing 19
  - 2.7.2. JBoss EAP 中的 MicroProfile OpenTracing 19
- 2.8. MICROPROFILE REST 客户端 20
  - 2.8.1. MicroProfile REST 客户端 20

第 3 章 在 JBOSS EAP 中管理 MICROPROFILE 22
- 3.1. MICROPROFILE OPENTRACING 管理 22
  - 3.1.1. 启用 MicroProfile Open Tracing 22
  - 3.1.2. 删除 microprofile-opentracing-smallrye 子系统 22
  - 3.1.3. 添加 microprofile-opentracing-smallrye 子系统 22
  - 3.1.4. 安装 Jaeger 23
- 3.2. MICROPROFILE 配置 23
  - 3.2.1. 在 ConfigSource 管理资源中添加属性 23
  - 3.2.2. 将目录配置为 ConfigSources 23
  - 3.2.3. 从 ConfigSource 类获取 ConfigSource 24
  - 3.2.4. 从 ConfigSourceProvider 类获取 ConfigSource 配置 24
- 3.3. MICROPROFILE 容错配置 25
  - 3.3.1. 添加 MicroProfile 容错扩展 25
- 3.4. MICROPROFILE 健康配置 25

3.4.1. 使用管理 CLI 检查健康状况 26
3.4.2. 使用管理控制台检查健康状况 26
3.4.3. 使用 HTTP 端点检查健康状况 26
3.4.4. 为 MicroProfile 健康启用身份验证 26
3.4.5. 决定服务器健康和就绪度的就绪度探测 27
3.4.6. 未定义探测时的全局状态 28
3.5. MICROPROFILE JWT 配置 28
3.5.1. 启用 microprofile-jwt-smallrye 子系统 28
3.6. MICROPROFILE 指标管理 29
3.6.1. 管理界面中提供的指标 29
3.6.2. 使用 HTTP 端点检查指标 29
3.6.3. 为 MicroProfile 指标 HTTP 端点启用身份验证 29
3.6.4. 获取 web 服务的请求数 30
3.7. MICROPROFILE OPENAPI 管理 31
3.7.1. 启用 MicroProfile OpenAPI 31
3.7.2. 使用 Accept HTTP 标头请求 MicroProfile OpenAPI 文档 31
3.7.3. 使用 HTTP 参数请求 MicroProfile OpenAPI 文档 32
3.7.4. 配置 JBoss EAP 以提供静态 OpenAPI 文档 32
3.7.5. 禁用 microprofile-openapi-smallrye 33
3.8. 单机服务器配置 33
3.8.1. 单机服务器配置文件 33
3.8.2. 使用 MicroProfile 子系统和扩展更新独立配置 35

第 4 章 为 JBOSS EAP 开发 MICROPROFILE 应用 36
4.1. MAVEN 和 JBOSS EAP MICROPROFILE MAVEN 存储库 36
4.1.1. 下载 JBoss EAP MicroProfile Maven 存储库补丁作为存档文件 36
4.1.2. 在您的本地系统中应用 JBoss EAP MicroProfile Maven 存储库补丁 36
4.1.3. 支持的 JBoss EAP MicroProfile BOM 37
4.1.4. 使用 JBoss EAP MicroProfile Maven 存储库 37
4.2. MICROPROFILE 配置开发 38
4.2.1. 为 MicroProfile 配置创建 Maven 项目 38
4.2.2. 在应用中使用 MicroProfile Config 属性 40
4.3. MICROPROFILE 容错应用开发 42
4.3.1. 添加 MicroProfile 容错扩展 42
4.3.2. 为 MicroProfile 容错配置 Maven 项目 42
4.3.3. 创建容错应用程序 43
4.4. MICROPROFILE 健康开发 46
4.4.1. 自定义健康检查示例 46
4.4.2. @Liveness 注释示例 47
4.4.3. @Readiness 注释示例 48
4.5. MICROPROFILE JWT 应用程序开发 48
4.5.1. 启用 microprofile-jwt-smallrye 子系统 48
4.5.2. 配置 Maven 项目以开发 JWT 应用程序 49
4.5.3. 使用 MicroProfile JWT 创建应用 50
4.6. MICROPROFILE 指标开发 54
4.6.1. 创建 MicroProfile 指标应用 54
4.7. 开发 MICROPROFILE OPENAPI 应用 56
4.7.1. 启用 MicroProfile OpenAPI 56
4.7.2. 为 MicroProfile OpenAPI 配置 Maven 项目 56
4.7.3. 创建 MicroProfile OpenAPI 应用 59
4.7.4. 配置 JBoss EAP 以提供静态 OpenAPI 文档 62
4.8. MICROPROFILE REST 客户端开发 63
4.8.1. MicroProfile REST 客户端与 Jakarta RESTful Web 服务语法的比较 63

4.8.2. MicroProfile REST 客户端中的提供程序编程注册 64
4.8.3. 在 MicroProfile REST 客户端中声明提供程序注册 64
4.8.4. MicroProfile REST 客户端中的标头声明规格 64
4.8.5. MicroProfile REST 客户端中的 ResponseExceptionMapper 65
4.8.6. 使用 MicroProfile REST 客户端进行上下文依赖项注入 65

第 5 章 在 OPENSHIFT 镜像上构建并运行 JBOSS EAP XP 的微服务应用程序 67
5.1. 为应用程序部署准备 OPENSHIFT 67
5.2. 配置 RED HAT CONTAINER REGISTRY 的身份验证 67
5.3. 为 JBOSS EAP XP 导入最新的 OPENSHIFT 镜像流和模板 68
5.4. 在 OPENSHIFT 上部署 JBOSS EAP XP SOURCE-TO-IMAGE(S2I)应用 69
5.5. 为 JBOSS EAP XP SOURCE-TO-IMAGE(S2I)应用完成部署后的任务 71

第 6 章 功能修剪 73
6.1. 可用的 JBOSS EAP 层 73
6.1.1. 基础层 73
datasources-web-server 73
jaxrs-server 74
cloud-server 74
6.1.2. decorator 层 74
ejb-lite 74
Jakarta Enterprise Beans 75
ejb-local-cache 75
ejb-dist-cache 75
jdr 75
Jakarta Persistence 75
jpa-distributed 76
Jakarta Server Faces 76
microprofile-platform 76
Observability（可观察性） 77
remote-activemq 77
sso 77
web-console 77
web-clustering 77
web-passivation 77
Web 服务 77

第 7 章 在红帽 CODEREADY STUDIO 上为 JBOSS EAP 启用 MICROPROFILE 应用开发 79
7.1. 配置 CODEREADY STUDIO 以使用 MICROPROFILE 功能 79
7.2. 将 MICROPROFILE 快速入门用于 CODEREADY STUDIO 80

第 8 章 可引导 JAR 82
8.1. 关于可引导 JAR 82
8.2. JBOSS EAP MAVEN 插件 82
8.3. 可引导 JAR 参数 83
8.4. 为您的可引导 JAR 服务器指定 GALLEON 层 84
8.5. 在 JBOSS EAP 裸机平台上使用可引导 JAR 86
8.6. 在 JBOSS EAP 裸机平台上创建 HOLLOW 可引导 JAR 89
8.7. CLI 脚本 90
8.8. 在 JBOSS EAP OPENSHIFT 平台上使用可引导 JAR 91
8.9. 为 OPENSHIFT 配置可引导 JAR 94
8.10. 在 OPENSHIFT 上的应用程序中使用 CONFIGMAP 95
8.11. 创建可引导 JAR MAVEN 项目 97
8.12. 为可引导 JAR 启用 JSON 日志记录 99

8.13. 为多个可引导 JAR 实例启用 WEB 会话数据存储 103
8.14. 使用 CLI 脚本为可引导 JAR 启用 HTTP 身份验证 108
8.15. 使用红帽单点登录保护 JBOSS EAP 可引导 JAR 应用程序 112
8.16. 在 DEV 模式下打包可引导 JAR 117
8.17. 将 JBOSS EAP 补丁应用到可引导 JAR 118

第 9 章 REFERENCE 120
9.1. MICROPROFILE 配置参考 120
9.1.1. 默认 MicroProfile 配置属性 120
9.1.2. MicroProfile Config SmallRye ConfigSources 120
9.2. MICROPROFILE 容错参考 120
9.2.1. MicroProfile 容错配置属性 120
9.3. MICROPROFILE JWT 参考 121
9.3.1. MicroProfile 配置 JWT 标准属性 121
9.4. MICROPROFILE OPENAPI 参考 121
9.4.1. MicroProfile OpenAPI 配置属性 121

使开源包含更多

红帽承诺替换我们的代码、文档和网页属性中存在问题的语言。我们从这四个术语开始： master、slave、blacklist 和 whitelist。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看 CTO Chris Wright 信息。

对红帽文档提供反馈

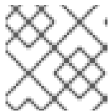
我们感谢您对我们文档的反馈。要提供反馈，您可以突出显示文档中的文本并添加注释。按照以下步骤了解提交对红帽文档的反馈。

先决条件

- 登录到红帽客户门户。
- 在红帽客户门户中，以多页 HTML 格式查看文档。

流程

1. 点 **Feedback** 查看现有的用户反馈信息。



注意

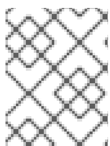
反馈功能仅在多页 HTML 格式中启用。

2. 高亮标记您要提供反馈的文档中的部分。
3. 在您选择的文本旁的提示菜单中，点 **Add Feedback**。
文本框将在页面右侧的"反馈"部分中打开。
4. 在文本框中输入您的反馈，然后点 **Submit**。
您已创建了文档问题。
5. 要查看问题，请单击反馈视图中的问题跟踪器链接。

第 1 章 适用于最新 MICROPROFILE 功能的 JBOSS EAP XP

1.1. 关于 JBOSS EAP XP

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

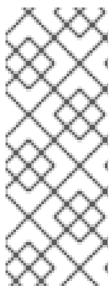


注意

JBoss EAP XP 受到单独的支持和生命周期政策约束。如需了解更多详细信息，请参阅 JBoss Enterprise Application Platform 扩展包支持和生命周期政策 页。

JBoss EAP XP 补丁提供以下 MicroProfile 4.0 组件：

- MicroProfile Config
- MicroProfile 容错
- MicroProfile Health
- MicroProfile JWT
- MicroProfile Metrics
- MicroProfile OpenAPI
- MicroProfile OpenTracing
- MicroProfile REST 客户端
- MicroProfile 被动消息传递



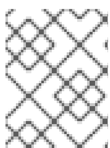
注意

MicroProfile 被动消息传递子系统支持红帽 AMQ 流。此功能实施 MicroProfile 主动消息传递 1.0 API，红帽提供作为 JBoss EAP XP 3.0.0 技术预览的功能。

红帽在 JBoss EAP 上测试了红帽 AMQ Streams 2021.Q2。但是，请查看红帽 JBoss 企业应用平台支持的配置页面，以获取有关已在 JBoss EAP XP 3.0.0 上测试的最新红帽 AMQ Streams 版本的信息。

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 3.0.x 补丁与 JBoss EAP 7.4 发行版兼容。



注意

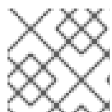
您可以通过 XP 管理器和 EAP 存档或使用 JBoss EAP XP OpenShift 容器镜像安装 JBoss EAP XP。您不能在 EAP RPM 基础上安装 JBoss EAP XP。

其它资源

- 有关在最新 JBoss EAP 版本上安装最新 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 3.0.0。

1.3. JBOSS EAP XP 管理器

JBoss EAP XP 管理器是可从 **产品下载页面** 下载的可执行 **jar** 文件。使用 JBoss EAP XP 管理器应用 JBoss EAP XP 补丁流中的 JBoss EAP XP 补丁程序。补丁中包含 MicroProfile 4.0 实施，以及这些 MicroProfile 4.0 实施的错误修复。

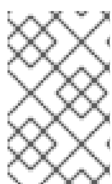


注意

您无法使用管理控制台管理 JBoss EAP XP 补丁。

如果您不带任何参数运行 JBoss EAP XP 管理器，或者使用 **help** 命令运行 JBoss EAP XP 管理器，则您会获得所有可用命令的列表，并包含其作用的说明。

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 服务器以管理 JBoss EAP XP 补丁流。如果要省略此设置，请在 **JBOSS_HOME** 环境变量中指定服务器的路径。**--jboss-home** 的优先级高于环境变量。

1.4. JBOSS EAP XP MANAGER 3.0 命令

JBoss EAP XP 管理器 3.0 提供了不同的命令来管理 JBoss EAP XP 补丁流并应用 JBoss EAP 7.4.x 基本补丁。

使用以下命令：

patch-apply

使用此命令将补丁应用到您的 JBoss EAP 安装。
patch-apply 命令类似于 **patch 应用** 管理 CLI 命令。**patch-apply** 命令仅接受使用 工具应用补丁所需的参数。它为其他 **patch 应用** 管理 CLI 命令参数使用默认值。

您可以使用 **patch-apply** 命令，将补丁应用到在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
```

当您应用 XP 补丁时，JBoss EAP XP Manager 3.0 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 3.0 补丁流的服务器上尝试安装 JBoss EAP XP 2.0 补丁会导致以下错误：

```
java.lang.IllegalStateException: The JBoss EAP XP patch stream in the patch 'jboss-eap-
xp-2.0' does not match the currently enabled JBoss EAP XP patch stream [jboss-eap-xp-
3.0]
 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)
```

- 尝试在尚未为 JBoss EAP XP 3.0 补丁流设置的服务器上安装 JBoss EAP XP 3.0 补丁会导致以下错误：

```
java.lang.IllegalStateException: You are attempting to install a patch for the 'jboss-eap-xp-3.0' 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
```

设置

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

- 启用 JBoss EAP XP 3.0 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。
  如果已安装旧配置文件，新文件将保存为目标配置目录中的时间戳副本，如 **standalone-microprofile-yyyMMdd-HHmms.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。

- 由于处于当前状态，因此任何支持策略更改。
- JBoss EAP XP 的主要版本。
- 启用补丁流及其累积补丁 ID。
- 用于更改状态的可用 JBoss EAP XP 管理器命令.

使用 status 命令的示例

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

upgrade

使用此命令将旧的 JBoss EAP XP 补丁流升级到 JBoss EAP 服务器中的最新补丁流。
使用 **upgrade 命令时**，JBoss EAP XP 管理器执行以下操作：

- 创建 文件的备份，在服务器中启用旧补丁流。
- 启用 JBoss EAP XP 3.0 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。如果已安装旧配置文件，新文件将保存为目标配置目录中的时间戳副本，如 **standalone-microprofile-yyyMMdd-HHmms.xml**。
- 如果出现问题，JBoss EAP XP 管理器会尝试从它创建的备份中恢复之前的补丁流。
  您可以使用 **--jboss-config-directory** 参数设置目标目录

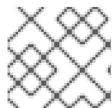
使用 upgrade 命令的示例：

```
$ 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 3.0.0

在 JBoss EAP 7.4 基础服务器上安装 JBoss EAP XP 3.0.0。

使用 JBoss EAP XP manager 3.0.0 管理 JBoss EAP XP 3.0.0 补丁流。



注意

JBoss EAP XP 3.0.0 认证 JBoss EAP 7.4.x。

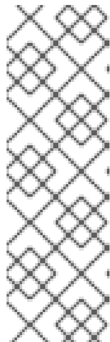
先决条件

- 您已从产品下载页面 **下载了** 以下文件：
  - **jboss-eap-xp-3.0.0-manager.jar** 文件(JBoss EAP XP Manager 3.0)
  - JBoss EAP 7.4 服务器存档文件
  - JBoss EAP XP 3.0.0 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. 使用以下命令将 JBoss EAP XP Manager 3.0.0 设置为管理 JBoss EAP XP 3.0 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 3.0.0 补丁。使用 **--xp-patch** 参数包括 JBoss EAP XP 3.0.0 补丁的路径。

例如：

```
$ java -jar jboss-eap-xp-manager.jar setup --jboss-home=<path_to_eap> --xp-patch <path_to_patch>jboss-eap-xp-3.0.0-patch.zip
```

服务器现在可以管理 JBoss EAP XP 3.0.0 补丁流。

3. 可选：如果您尚未使用 **--xp-patch** 参数将 JBoss EAP 3.0.0 补丁应用到 JBoss EAP 服务器，请使用 JBoss EAP XP Manager 3.0.0 命令应用 JBoss EAP XP 3.0.0 补丁 ：

```
$ java -jar jboss-eap-xp-manager.jar patch-apply --jboss-home=<path_to_eap> --patch=<path_to_patch>jboss-eap-xp-3.0.0-patch.zip
```

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

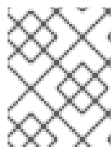
在您使用 JBoss EAP XP 3.0.0 修补程序修补 JBoss EAP 服务器时，JBoss EAP 服务器现在可以管理 JBoss EAP XP 3.0.0 补丁流。

其它资源

- JBoss EAP XP Manager 3.0 命令

1.6. 卸载 JBOSS EAP XP

卸载 JBoss EAP XP 将移除与启用 JBoss EAP XP 3.0.0 补丁流和 MicroProfile 4.0 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 补丁流的配置文件。

流程

- 发出以下命令卸载 JBoss EAP XP 3.0.0：

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

要再次安装 MicroProfile 4.0 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 4.0 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。
- 由于处于当前状态，因此任何支持策略更改。
- JBoss EAP XP 的主要版本。
- 启用补丁流及其累积补丁 ID。
- 用于更改状态的可用 JBoss EAP XP 管理器命令.

JBoss EAP XP 可以处于以下状态之一：

未设置

JBoss EAP 是干净的，未设置 JBoss EAP XP。

设置

JBoss EAP 设置了 JBoss EAP XP。不会显示 XP 补丁流的版本，因为用户可以使用 CLI 来确定修补程序。

不一致

与 JBoss EAP XP 相关的文件处于不一致的状态。这是错误条件，不应正常发生。如果您遇到此错误，请删除 JBoss EAP XP 管理器（如卸载 JBoss EAP XP 主题中所述），然后使用 **setup** 命令再次安装 JBoss EAP XP。

流程

- 发出以下命令来查看 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 3.0.0

1.8. 回滚 JBOSS EAP XP 和 JBOSS EAP 7.4.X 基本补丁

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 基本补丁。

其他资源

- 有关回滚 JBoss EAP XP 补丁或 JBoss EAP 7.4.x 基本补丁的更多信息，请参阅使用 管理 CLI 滚动补丁。

第 2 章 了解 MICROPROFILE

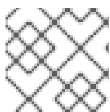
2.1. MICROPROFILE CONFIG

2.1.1. JBoss EAP 中的 MicroProfile 配置

配置数据可以动态变化，应用需要能够在不重新启动服务器的情况下访问最新的配置信息。

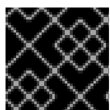
MicroProfile 配置提供可移植的外部化配置数据。这意味着，您可以将应用和微服务配置为在多个环境中运行，无需修改或重新打包。

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



注意

MicroProfile Config 仅在 JBoss EAP XP 中受支持。JBoss EAP 不支持它。



重要

如果您要添加自己的配置实施，则需要在最新版本的配置界面中使用方法。

其它资源

- MicroProfile Config
- SmallRye Config
- 配置实现

2.1.2. MicroProfile 配置中支持的 MicroProfile 配置源

MicroProfile 配置属性可以来自不同的位置，可以是不同的格式。这些属性由 ConfigSources 提供。ConfigSources 是 **org.eclipse.microprofile.config.spi.ConfigSource** 接口的实施。

MicroProfile 配置规范提供以下默认 **ConfigSource** 实施来检索配置值：

- **System.getProperties()**.
- **System.getenv()**.
- 课程路径 **上的所有 META-INF/microprofile-config.properties** 文件.

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

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

2.2. MICROPROFILE 容错

2.2.1. 关于 MicroProfile 容错规范

MicroProfile 容错规范定义策略，以处理分布式微服务中固有的错误。

MicroProfile 容错规范定义以下策略来处理错误：

Timeout
定义执行必须完成的时间长度。定义超时可防止无限期等待执行。

Retry
定义用于重试失败执行的条件。

回退
在执行失败时提供替代方案。

CircuitBreaker
在临时停止之前，定义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。

舱壁
隔离系统部分内的故障，以便系统的其余部分仍能正常工作。

异步
在单独的线程中执行客户端请求。

其它资源

- MicroProfile 容错规范

2.2.2. JBoss EAP 中的 MicroProfile 容错

The **microprofile-fault-tolerance-smallrye** 子系统为 JBoss EAP 中的 MicroProfile 容错提供支持。该子系统仅可在 JBoss EAP XP 流中使用。

The **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。绑定至类的注释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别的拦截器绑定，则应用以下限制：
  - 不得宣布课程结束。
  - 类不得包含任何静态、私有或最终方法。
- 如果组件类的非静态、非专用方法声明了方法级拦截器绑定，则方法或组件类不能最终声明。

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 Contexts 和 Dependency Injection 规范中定义的业务方法调用。
- 如果满足以下两个条件，则操作被视为容错：
  - 该方法本身不与任何容错拦截器绑定。
  - 包含该方法的类不绑定到任何容错拦截器。

除了 MicroProfile 容错提供的配置选项之外，**microprofile-fault-tolerance-smallrye** 子系统还提供以下配置选项：

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

其它资源

- MicroProfile 容错规范
- SmallRye Fault Tolerance 项目

2.3. MICROPROFILE HEALTH

2.3.1. JBoss EAP 中的 MicroProfile 健康

JBoss EAP 包含 SmallRye 健康组件，可用于确定 JBoss EAP 实例是否按预期响应。此功能默认为启用。

MicroProfile 健康仅在将 JBoss EAP 作为单机服务器运行时才可用。

MicroProfile 健康规范定义以下健康检查：

就绪

确定应用是否已准备好处理请求。注释 **@Readiness** 提供此健康检查。

存活度

确定应用是否正在运行。注释 **@Liveness** 提供此健康检查。

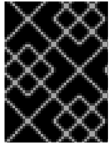
MicroProfile Health 3.0 中移除了 **@Health** 注释。

MicroProfile 健康 3.0 有以下破坏更改：

- 修剪 **@Health** 限定符

- 将 HealthCheckResponse **state** 参数重命名为 **status**，以修复降序问题。这也导致相应方法重命名。

如需有关 MicroProfile Health 3.0 中破坏更改的更多信息，请参阅 MicroProfile Health 3.0 发行说明。



重要

:empty-readiness-checks-status 和 **:empty-liveness-checks-status** 管理属性在未定义 就绪 **度 或 存活度** 探测时指定全局状态。

其它资源

- 未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

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

下列功能由 **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.

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 部署不依赖于像传统的 JBoss EAP 部署一样管理 SecurityDomain 资源。相反，将创建虚拟 SecurityDomain 并在 MicroProfile JWT 部署中使用。

由于 MicroProfile JWT 部署完全在 MicroProfile Config 属性和 **microprofile-jwt-smallrye** 子系统内配置，虚拟 SecurityDomain 不需要任何其他管理的配置来进行部署。

2.4.3. JBoss EAP 中的 MicroProfile JWT 激活

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

MicroProfile JWT 集成通过以下方式为应用激活：

- 作为部署流程的一部分，JBoss EAP 扫描应用存档，了解是否存在 **auth-method**。
- 如果存在 **auth-method**，并且定义为 **MP-JWT**，则将激活 MicroProfile JWT 集成。

auth-method 可以在以下任一文件中指定：

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

如果在类中都定义了 **auth-method**，使用注释，并在 web.xml 配置文件中定义了，则使用 **web.xml** 配置文件中的定义。

2.4.4. JBoss EAP 中 MicroProfile JWT 的限制

JBoss EAP 中的 MicroProfile JWT 实施有一些限制。

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

- MicroProfile JWT 实施仅解析来自 **mp.jwt.verify.publickey** 属性中提供的 JSON Web 密钥集 (JWKS)的第一个密钥。因此，如果在第二个密钥后通过第二个密钥或密钥签名令牌声明，则令牌将失败，并且包含令牌的请求不会被授权。
- 不支持 JWKS 的 Base64 编码。

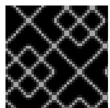
在这两种情况下，都可以引用明文 JWKS，而不使用 **mp.jwt.verify.publickey.location** 配置属性。

2.5. MICROPROFILE METRICS

2.5.1. JBoss EAP 中的 MicroProfile 指标

JBoss EAP 包括 SmallRye 指标组件。SmallRye 指标组件利用 **microprofile-metrics-smallrye** 子系统提供 MicroProfile 指标功能。

The **microprofile-metrics-smallrye** 子系统提供 JBoss EAP 实例的监控数据。子系统默认为启用。



重要

The **microprofile-metrics-smallrye** 子系统仅在单机配置中启用。

其它资源

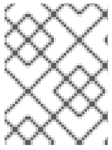
- SmallRye Metrics
- MicroProfile Metrics

2.6. MICROPROFILE OPENAPI

2.6.1. JBoss EAP 中的 MicroProfile OpenAPI

MicroProfile OpenAPI 使用 **microprofile-openapi-smallrye** 子系统集成在 JBoss EAP 中。

MicroProfile OpenAPI 规范定义一个 HTTP 端点，提供 OpenAPI 3.0 文档。OpenAPI 3.0 文档描述了主机的 REST 服务。OpenAPI 端点使用配置的路径进行注册，如 http://localhost:8080/openapi，它属于与部署关联的主机的根目录。



注意

目前，虚拟主机的 OpenAPI 端点只能记录单个部署。若要将 OpenAPI 与注册在同一虚拟主机的不同上下文路径的多个部署一起使用，每一部署必须使用不同的端点路径。

默认情况下，OpenAPI 端点会返回 YAML 文档。您还可以使用 Accept HTTP 标头或格式查询参数来请求 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 文档也可以使用 HTTPS 使用。例如，HTTPS 的端点是 https://localhost:8443/openapi。

2.7. MICROPROFILE OPENTRACING

2.7.1. MicroProfile OpenTracing

跟踪服务边界请求的能力非常重要，特别是在请求可以在其生命周期内穿过多个服务的微服务环境中。

MicroProfile OpenTracing 规范定义用于访问 CDI-bean 应用中 OpenTracing 兼容 **Tracer** 接口的行为和 API。**Tracer** 接口自动跟踪 JAX-RS 应用。

行为指定如何为传入和传出请求自动创建 OpenTracing Spans。API 定义如何显式禁用或启用给定端点的追踪。

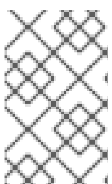
其它资源

- 有关 MicroProfile OpenTracing 规范的更多信息，请参见 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。

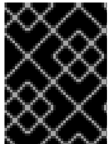
MicroProfile OpenTracing 2.0 支持应用的追踪请求。您可以使用管理 CLI 或管理控制台 JBoss EAP 管理 API，为 Jakarta EE 中常用的组件配置默认的 Jaeger Java 客户端跟踪器，以及一组检测库。



注意

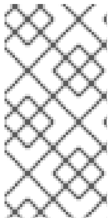
部署到 JBoss EAP 服务器的每个单独的 WAR 都会自动拥有自己的 **跟踪器** 实例。EAR 中的每个 WAR 都被视为单独的 WAR，每个 WAR 都有自己的 **Tracer** 实例。默认情况下，用于 Jaeger Client 的服务名称派生自部署名称，通常是 WAR 文件名。

在 **microprofile-opentracing-smallrye** 子系统中，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 客户端跟踪器作为技术预览提供。附属于 Jaeger 客户端跟踪器的系统属性和环境变量可能会在以后的版本中有所改变，并相互不兼容。



注意

默认情况下，Java Jaeger Client for Java 的 probabilistic 抽样策略被设置为 **0.001**，这意味着只对大约一千个 trace 进行抽样。若要对每个请求进行示例，请将系统属性 **JAEGER_SAMPLER_TYPE** 设置为 **const**，将 **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 客户端。
- 有关 **Tracer** 接口的更多信息，请参阅 **Tracer** javadoc。
- 有关覆盖默认追踪器和追踪 Jakarta 上下文和依赖注入 Bean 的更多信息，请参阅《*开发指南*》中使用 Eclipse MicroProfile OpenTracing to Trace 请求 。
- 有关配置 Jaeger 客户端的更多信息，请参阅 Jaeger 文档。
- 如需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 3.0.0 支持 MicroProfile REST 客户端 2.0，它构建于 Jakarta RESTful Web Services 2.1.6 客户端 API，以提供通过 HTTP 调用 RESTful 服务的类型安全方法。MicroProfile 类型 Safe REST 客户端定义为 Java 接口。利用 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 客户端使用以下功能：

- 直观的语法
- 供应商的程序注册
- 供应商声明注册
- 标头声明规格
- 在服务器中传播标头
- **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 服务语法之间的比较
- MicroProfile REST 客户端中的提供程序编程注册
- 在 MicroProfile REST 客户端中声明提供程序注册
- MicroProfile REST 客户端中的标头声明规格
- 在 MicroProfile REST 客户端的服务器上传播标头
- 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

第 3 章 在 JBOSS EAP 中管理 MICROPROFILE

3.1. MICROPROFILE OPENTRACING 管理

3.1.1. 启用 MicroProfile Open Tracing

使用下列管理 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

The **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。如果您在启用了 MicroProfile OpenTracing 时遇到系统内存或性能下降的问题，您可能想要禁用 **microprofile-opentracing-smallrye** 子系统。

您可以在管理 CLI **中使用 remove** 操作，在全局范围内为给定服务器禁用 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. 添加 microprofile-opentracing-smallrye 子系统

您可以通过添加 **microprofile-opentracing-smallrye** 子系统来启用microprofile-opentracing-smallrye 子系统。在管理 CLI 中使用 **add** 操作，为给定的服务器全局启用 MicroProfile OpenTracing 功能。

流程

1. 添加 子系统。

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

2. 重新加载服务器以使更改生效。

```
reload
```

3.1.4. 安装 Jaeger

使用 **docker** 安装 Jaeger。

先决条件

- **已安装** Docker。

流程

1. 使用 **docker** 在 CLI 中运行以下命令来安装 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

3.2.1. 在 ConfigSource 管理资源中添加属性

您可以将属性直接作为管理资源存储在 **config-source** 子系统中。

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

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. 创建一个文件名 **来存储** 属性 **名称** 的值：

```
$ touch name
```

4. 在文件中添加 属性值：

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

5. 创建一个 ConfigSource，其中的文件名是属性，文件内容为属性的值：

```
/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

您可以创建和配置自定义 **org.eclipse.microprofile.config.spi.ConfigSource** 实施类，以提供配置值的来源。

流程

- 以下管理 CLI 命令为名为 **org.example.MyConfig Source 的实施类创建一个 Config Source,** 该类由名为 **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})
  ```

  此命令将为 **microprofile-config-smallrye** 子系统生成以下 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>
  ```

由自定义 **org.eclipse.microprofile.config.spi.ConfigSource** 实施类提供的属性可用于任何 JBoss EAP 部署。

3.2.4. 从 ConfigSourceProvider 类获取 ConfigSource 配置

您可以创建和配置自定义 **org.eclipse.microprofile.config.spi.ConfigSourceProvider** 实施类，以注册多个 **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** 的实现类创建一个 **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** 文件中。

此命令为 **microprofile-config-smallrye** 子系统生成以下 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 实施提供的属性可用于任何 JBoss EAP 部署。

其他资源

- 有关如何将全局模块添加到 JBoss EAP 服务器的信息，请参阅 JBoss EAP *配置指南中的定义* 全局模块。

3.3. MICROPROFILE 容错配置

3.3.1. 添加 MicroProfile 容错扩展

作为 JBoss EAP XP 的一部分提供的 MicroProfile 容错扩展包含在 **standalone-microprofile.xml 和 standalone-microprofile-ha.xml** 配置中。

扩展不包含在标准 **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 包。

流程

1. 使用以下命令添加 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 健康配置

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运行状况。

检查运行时操作显示健康检查和全局结果作为布尔值。

流程

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

3.4.3. 使用 HTTP 端点检查健康状况

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

/health 端点的默认地址可从管理接口访问，即 **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
```

访问此上下文时，将以 JSON 格式显示健康检查，指示服务器是否正常运行。

3.4.4. 为 MicroProfile 健康启用身份验证

您可以将 **健康** 上下文配置为需要进行身份验证才能访问。

流程

1. 在 **microprofile -health-smallrye 子系统上，将 security- enabled** 属性设为 **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
```

访问 **/health** 端点的任何后续尝试都会触发身份验证提示符。

3.4.5. 决定服务器健康和就绪度的就绪度探测

JBoss EAP XP 3.0.0 支持三种就绪度探测，以确定服务器健康和就绪度。

- **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**。

这些就绪度探测会被默认启用。您可以使用 MicroProfile Config property **mp.health.disable-default-procedures 来禁用探测。**

以下示例演示了将三个探测与 **检查操作结合使用** ：

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

其他资源

- JBoss EAP 中的 MicroProfile 健康

- 未定义探测时的全局状态

3.4.6. 未定义探测时的全局状态

:empty-readiness-checks-status 和 **:empty-liveness-checks-status** 管理属性在未定义就绪 **度** **或** **存活度** 探测时指定全局状态。

这些属性允许应用报告"DOWN"，直到探测验证应用程序是否已就绪或存活为止。默认情况下，应用程序报告"UP"。

- 如果没有定义 **就绪度探测**，**: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}"
}
```

- 如果没有定义 **存活** 度探测，**: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}"
}
```

  检查 **就绪** **度和存活度** 探测的 **/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，您可以按照如下所示进行添加：

先决条件

- 已安装 EAP XP。

流程

1. 在 JBoss EAP 中启用 MicroProfile JWT smallrye 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
```

The **microprofile-jwt-smallrye** 子系统已启用。

3.6. MICROPROFILE 指标管理

3.6.1. 管理界面中提供的指标

JBoss EAP 子系统指标以 Prometheus 格式公开。

指标在 JBoss EAP 管理界面中自动可用，包含以下上下文：

- **/metrics/** - 包含 MicroProfile 3.0 规范中指定的指标。
- **/metrics/vendor** - 包含特定于供应商的指标，如内存池。
- **/metrics/application** - 包含来自使用 MicroProfile 指标 API 的部署和子系统的指标。

指标名称基于子系统和属性名称。例如，**undertow** 子系统为应用部署中的每个 servlet 公开指标属性 **request-count**。此指标的名称为 **jboss_undertow_request_count**。前缀 **jboss** 将 JBoss EAP 识别为指标的来源。

3.6.2. 使用 HTTP 端点检查指标

使用 HTTP 端点，检查 JBoss EAP 管理接口上可用的指标。

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，以要求用户获得授权才能访问该上下文。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** 子系统上，将 **security- enabled** 属性设为 **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
   ```

之后任何访问 **指标** 端点的尝试都会产生身份验证提示符。

3.6.4. 获取 web 服务的请求数

获取公开其请求数指标的 Web 服务的请求数。

以下流程使用 **helloworld-rs** faststart 作为 Web 服务来获取请求计数。此快速入门可从 下载快速入门来源： 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：
   - 在启用了统计信息的情况下启动单机服务器：

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

   - 对于已经运行的服务器，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** 快速入门：
   - 在快速启动的根目录中，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，并为 **request_count** 过滤：

   ```
   $ 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
   ```

   返回的属性值为 **0.0**。

4. 在 Web 浏览器中访问位于 http://localhost:8080/helloworld-rs/ 的快速入门，再单击任何链接。

5. 再次从 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
```

该值更新至 **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

The **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，您也可遵循 使用 MicroProfile 子系统和扩展来更新 **standalone.xml** 配置文件的步骤。

流程

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

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

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

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

3. 重新加载服务器：

```
reload
```

启用 The **microprofile-openapi-smallrye** 子系统。

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 MicroProfile OpenAPI 文档（JSON 格式）。

默认情况下，OpenAPI 端点返回 YAML 文档。

先决条件

- 正在查询的部署配置为返回 MicroProfile OpenAPI 文档。

流程

- 发出以下 **curl** 命令以查询 **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，从部署请求请求 MicroProfile OpenAPI 文档（JSON 格式）。

默认情况下，OpenAPI 端点返回 YAML 文档。

先决条件

- 正在查询的部署配置为返回 MicroProfile OpenAPI 文档。

流程

- 发出以下 **curl** 命令以查询 **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** 表示将返回 JSON 文档。

3.7.4. 配置 JBoss EAP 以提供静态 OpenAPI 文档

配置 JBoss EAP 以提供静态 OpenAPI 文档，描述主机的 REST 服务。

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 注释之前处理静态 OpenAPI 文档。

在生产环境中，在提供静态文档时禁用注释处理。禁用注解处理可确保客户端可以使用不可变和版本化的 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 是含有应用的 **pom.xml** 配置文件的目录。

2. 查询 OpenAPI 端点，将输出重定向到文件中：

```
$ 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. 重新构建应用程序：

```
$ 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. 单机服务器配置

3.8.1. 单机服务器配置文件

JBoss EAP XP 包含其他单机服务器配置文件 **standalone-microprofile.xml** 和 **standalone-microprofile-ha.xml**。

JBoss EAP 附带的标准配置文件保持不变。请注意，JBoss EAP XP 3.0.0 不支持使用 **domain.xml** 文件或域模式。

表 3.1. JBoss EAP XP 中提供的独立配置文件.

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。	包含有关服务器的信息，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。

配置文件	用途	包括的功能	排除的功能
standalone-microprofile.xml	此配置文件支持使用 MicroProfile 的应用。	包含有关服务器的信息，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.
standalone-microprofile-ha.xml	此独立文件支持使用 MicroProfile 的应用。	除了默认的子系统外，还包括 **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.
standalone-full.xml		除了默认的子系统外，还包括 **messaging-activemq** 和 **iiop-openjdk** 子系统。	
standalone-full-ha.xml	支持每一种可能的子系统.	包括消息传递子系统和高可用性子系统，以及默认子系统。	
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必需的最小子系统。		

默认情况下，将 JBoss EAP 作为单机服务器启动使用 **standalone.xml** 文件。要使用单机 MicroProfile 配置启动 JBoss EAP，请使用 **-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

3.8.2. 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** 脚本通过 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，而非自定义配置。

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添加下列 MicroProfile 子系统和扩展（如果它们不存在）：

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

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

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单机服务器配置：

```
$ 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

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，提取到现有的红帽 JBoss 企业应用平台 7.4.0.GA Maven 存储库中。增量归档文件不会覆盖或删除任何现有的文件，因此无需回滚。

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 **红帽 JBoss 企业应用平台** 条目并进行选择。
4. 从 **产品** 下拉列表中，选择 **JBoss EAP XP**。
5. 从 **Version** 下拉列表中，选择 **2.0.0**。
6. 单击 **Releases** 选项卡。
7. 在列表中找到 **JBoss EAP XP 3.0.0 Incremental Maven Repository** 然后单击 **Download**。
8. 将归档文件保存到您的本地目录。

其它资源

- 若要了解更多有关 JBoss EAP Maven 存储库的信息，请参阅 JBoss EAP *开发指南中的关于* Maven 存储库。

4.1.2. 在您的本地系统中应用 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。

当您以增量归档文件的形式将补丁应用到存储库中时，新文件将添加到此存储库。增量归档文件不会覆盖或删除存储库中的任何现有文件，因此无需回滚。

先决条件

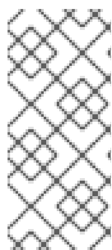
- 您已 下载并安装了 本地系统上的 Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven 存储库。
    - 检查您是否已在本地系统中安装了红帽 JBoss 企业应用平台 7.4 Maven 存储库的这一次要版本。
- 您已下载了您本地系统上的 JBoss EAP XP 2.0.0 增加 Maven 存储库。

流程

1. 查找红帽 JBoss 企业应用平台 7.4.0.GA Maven 存储库的路径。例如，**/path/to/repo/jboss-eap-7.3.0.GA-maven-repository/maven-repository/**。

2. 将下载的 JBoss EAP XP 2.0.0 Incremental Maven 存储库直接提取到红帽 JBoss 企业应用平台 7.4.0.GA Maven 存储库的 目录中。例如，打开一个终端并运行以下命令，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-3.0.0-incremental-maven-repository.zip -d
EAP_MAVEN_REPOSITORY_PATH
```



注意

EAP_MAVEN_REPOSITORY_PATH 指向 **jboss-eap-7.3.0.GA-maven-repository**。例如，此流程演示了路径 **/path/to/repo/jboss-eap-7.3.0.GA-maven-repository/** 的使用。

将 JBoss EAP XP Incremental Maven 存储库提取到红帽 JBoss 企业应用平台 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* 确定 JBoss EAP Maven 存储库的 URL。

4.1.3. 支持的 JBoss EAP MicroProfile BOM

JBoss EAP XP 3.0.0 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 是 **org.jboss.bom**，它打包了许多 JBoss EAP MicroProfile 支持的 API 依赖项，如 **microprofile-openapi-api** 和 **microprofile-config-api**。如果使用此 BOM，您不需要为受支持的 API 依赖项指定版本，因为 **jboss-eap-xp-microprofile** BOM 为依赖项指定此值。

4.1.4. 使用 JBoss EAP MicroProfile Maven 存储库

安装红帽 JBoss 企业应用平台 7.4.0.GA Maven 存储库并应用 JBoss EAP XP Incremental Maven 存储库后，您可以访问 **jboss-eap-xp-microprofile** BOM。然后，存储库名称变为 JBoss EAP MicroProfile Maven 存储库。BOM 在 JBoss EAP XP 增加 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的控制和可管理性。

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，覆盖配置的项目的全局和用户 Maven 设置。

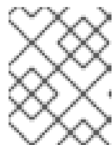
先决条件

- 您已在本地系统上安装了红帽 JBoss 企业应用平台 7.4 Maven 存储库，并且已将 JBoss EAP XP 增加 Maven 存储库应用到该存储库。

流程

1. 选择配置方法并配置 JBoss EAP MicroProfile Maven 存储库。
2. 在您配置了 JBoss EAP MicroProfile Maven 存储库后，将 **jboss-eap-xp-microprofile** BOM 添加到 POM 项目。以下示例演示了如何在 **pom.xml** 文件的 **<dependencyManagement>** 部分中配置 BOM：

```
<dependencyManagement>
  <dependencies>
    ...
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-xp-microprofile</artifactId>
      <version>3.0.0.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 为该元素指定一个 **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 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 Maven 项目，其中包含必要的依赖项，以及用于创建 MicroProfile 配置应用的目录结构。

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 **jboss-eap-xp-microprofile** BOM 中的 MicroProfile Config 构件和 MicroProfile REST 客户端构件的版本，请将 BOM 导入到项目 POM 文件的 **<dependencyManagement> 部分中**。

```
<dependencyManagement>
  <dependencies>
    <!-- importing the microprofile BOM adds MicroProfile specs -->
    <dependency>
        <groupId>org.jboss.bom</groupId>
        <artifactId>jboss-eap-xp-microprofile</artifactId>
        <version>3.0.0.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 POM 文件的 **<dependency> 部分**。以下示例演示了将 MicroProfile Config 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 Config。
- 安装了最新的 POM。
- Maven 项目已配置为创建 MicroProfile 配置应用。

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*** 是含有应用的 **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/
```

   在此目录中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，其包含以下内容：

```
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，其包含以下内容：

```
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 属性通过注释 **@ConfigProperty(name="name", defaultValue="jim")** 注入到类中。如果没有配置 **ConfigSource**，则返回 value **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*** 是含有应用的 **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*** 是含有应用的 **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 容错应用开发

4.3.1. 添加 MicroProfile 容错扩展

作为 JBoss EAP XP 的一部分提供的 MicroProfile 容错扩展包含在 **standalone-microprofile.xml 和 standalone-microprofile-ha.xml** 配置中。

扩展不包含在标准 **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 包。

流程

1. 使用以下命令添加 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 容错配置 Maven 项目

创建一个 Maven 项目，其中包含必要的依赖项，以及用于创建 MicroProfile 容错应用的目录结构。

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
```

命令可为项目和 **pom.xml** 配置文件创建目录结构。

2. 要让 POM 文件自动管理 **jboss-eap-xp-microprofile** BOM 中的 MicroProfile 容错构件的版本，请将 BOM 导入到项目 POM 文件的 **<dependencyManagement> 部分中**。

```
<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 构件添加到 POM 文件的 **<dependency> 部分**。以下示例演示了将 MicroProfile 容错依赖项添加到该文件：

```
<!-- 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

创建容错应用，实现容错模式的重试、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 是含有应用的 **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. 创建一个代表 Coffee.java 的 **Coffee.java** 的简单实体，其中包含以下内容：

```
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. 创建一个 Jakarta 上下文和依赖注入 Bean 作为 **CoffeeRepositoryService.java**，包含以下内容：

```
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. 定义重试到 **4** 的次数。
2. 以毫秒为单位定义超时间隔。
3. 定义在调用失败时要调用的回退方法。

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将执行基本的健康检查。如需更多详细信息，可以包括服务器或应用程序状态上的自定义健康检查。在运行时会自动发现和调用任何包含 **org.eclipse.microprofile.health.Liveness 注解** 或类级别的

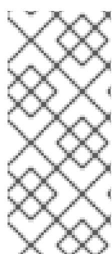
org.eclipse.microprofile.health.Readiness 注解的任何 Jakarta Contexts 和 Dependency Injection bean。

以下示例演示了如何创建返回 **UP** 状态的健康检查的新实施。

```
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();
    }
}
```

部署后，任何后续健康检查查询都会包括自定义检查，如下例中所示。

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



注意

您可以使用以下命令进行存活度和就绪度检查：

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

4.4.2. @Liveness 注释示例

以下是在应用中使用 **@Liveness** 注释的示例：

```
@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");
        }
    }
}
```

4.5. 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，您可以按照如下所示进行添加：

先决条件

- 已安装 EAP XP。

流程

1. 在 JBoss EAP 中启用 MicroProfile JWT smallrye 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
```

The **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 Maven 项目，其中包含必要的依赖项和用于开发 JWT 应用的目录结构。

先决条件

- 已安装 Maven。
- 启用了 **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
```

命令可为项目和 **pom.xml** 配置文件创建目录结构。

2. 要让 POM 文件自动管理 **jboss-eap-xp-microprofile** BOM 中的 MicroProfile JWT 构件的版本，请将 BOM 导入到项目 POM 文件的 **<dependencyManagement>** 部分中。

```
<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>
```

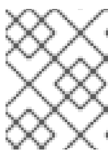
将 ${version.microprofile.bom} 替换为安装的 BOM 版本。

3. 将由 BOM 管理的 MicroProfile JWT 构件添加到 POM 文件的 **<dependency> 部分**。以下示例演示了将 MicroProfile JWT 依赖项添加到该文件中：

```
<!-- 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

创建应用，以基于 JWT 令牌对请求进行身份验证，并根据令牌持有者的身份实施授权。



注意

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

先决条件

- Maven 项目配置了正确的依赖项。

流程

1. 创建令牌生成器。
   此步骤用作参考。在生产环境中，请使用红帽 SSO 等身份提供程序。

   a. 创建生成器实用程序的令牌目录 **src/test/java** 目录并导航到此目录：

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

   b. 创建包含以下内容的类文件 **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，其中包含以下内容：

```
<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. 创建包含以下内容的类文件 **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.";
```

```
    }
   }
```

注释有 **@Path** 的方法是 Jakarta RESTful Web 服务端点。

注释 **@Claim** 定义 JWT 声明。

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

```
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")** 在部署期间启用 JWT RBAC。

5. 使用以下 Maven 命令编译应用程序：

```
$ mvn package
```

6. 使用令牌生成器实用程序生成 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

创建一个应用，它将向应用发出的请求数返回。

流程

1. 创建包含以下内容的类文件 **HelloService.java** ：

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

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

2. 创建包含以下内容的类文件 **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="helloworld-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 中重新执行以下命令：

```
$ 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="helloworld-rs-metrics.war",microprofile_scope="vendor"} 1.0
```

4.7. 开发 MICROPROFILE OPENAPI 应用

4.7.1. 启用 MicroProfile OpenAPI

The **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，您也可遵循 使用 MicroProfile 子系统和扩展来更新 **standalone.xml** 配置文件的步骤。

流程

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

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

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

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

3. 重新加载服务器：

```
reload
```

启用 The **microprofile-openapi-smallrye** 子系统。

4.7.2. 为 MicroProfile OpenAPI 配置 Maven 项目

创建 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。
- 配置了 JBoss EAP Maven 存储库。

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
```

命令可为项目和 **pom.xml** 配置文件创建目录结构。

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>3.0.0.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。

先决条件

- Maven 项目已配置为创建 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 是含有应用的 **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/
```

必须在此目录中创建下列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. 创建包含以下内容的类文件 **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

- 如需 MicroProfile SmallRye OpenAPI 中定义的注释列表，请参阅 MicroProfile OpenAPI 注释。

4.7.4. 配置 JBoss EAP 以提供静态 OpenAPI 文档

配置 JBoss EAP 以提供静态 OpenAPI 文档，描述主机的 REST 服务。

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 注释之前处理静态 OpenAPI 文档。

在生产环境中，在提供静态文档时禁用注释处理。禁用注解处理可确保客户端可以使用不可变和版本化的 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 是含有应用的 **pom.xml** 配置文件的目录。

2. 查询 OpenAPI 端点，将输出重定向到文件中：

   ```
   $ 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. 重新构建应用程序：

   ```
   $ 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 客户端与 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 远程方法调用(RMI)、JBoss 远程方法传递项目和 RESTEasy 中实施。例如，考虑资源：

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

以下示例演示了如何使用 Jakarta RESTful Web Services-native 来访问 **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，让 TestResource 类的调用 **在使用 TestResource Intf** 类时变得更容易，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")** 方法会产生 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. MicroProfile REST 客户端中的提供程序编程注册

借助 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. 在 MicroProfile REST 客户端中声明提供程序注册

使用 MicroProfile REST 客户端将 **org.eclipse.microprofile.rest.client.annotation.RegisterProvider** 注释添加到目标接口，如以下示例所示：

```
@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：

- 通过标注其中一个资源方法参数：

- 声明使用 **org.eclipse.microprofile.rest.client.annotation.ClientHeaderParam** 注释。

以下示例演示了通过使用注释 **@HeaderParam** 为其中一个资源方法参数添加注解来设置标头：

```
@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** 类的客户端一侧，它在 Jakarta RESTful Web Services 中定义。**ExceptionMapper.toResponse（）** 方法将服务器侧处理期间引发的 **Exception** 类转变为 **Response** 类。**ResponseExceptionMapper.toThrowable（）** 方法将客户端上收到的 **Response** 类和 HTTP 错误状态转换为 **Exception** 类。

您可以以编程或声明方式注册 **ResponseExceptionMapper** 类。如果没有注册的 **ResponseExceptionMapper** 类，默认的 **ResponseExceptionMapper** 类会将任何响应 **映射到** 一个 **WebApplicationException** 类。

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

利用 MicroProfile REST 客户端，您必须通过 **@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);
}
```

此处，MicroProfile REST 客户端实施为 **TestDataBase** 类服务创建一个客户端，让 **TestResourceImpl** 类能够轻松访问。但是，它不包括有关 **TestDataBase** 类实施路径的信息。此信息可以由可选的 **@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 **TestDataBase 的实现，地址** 为 https://localhost:8080/webapp。您还可以使用 MicroProfile 配置在外部提供信息：

```
<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.bluemondiamond.TestDatabase** 类的实施：

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

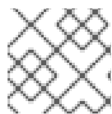
您可以向 Jakarta Contexts 和 Dependency Injection 客户端提供许多其他属性。例如，**com.mycompany.remoteServices.MyServiceClient/mp-rest/providers**，以逗号分隔的完全限定提供程序类名称列表，包含在客户端中。

其他资源

- 如需有关 MicroProfile REST 客户端规范的更多信息，请参阅 MicroProfile 的 Rest 客户端。
- 如需有关 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 OPENSHIFT 镜像上构建并运行 JBOSS EAP XP 的微服务应用程序

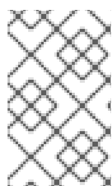
您可以在 JBoss EAP XP 的 OpenShift 镜像上构建并运行微服务应用。



注意

JBoss EAP XP 仅支持在 OpenShift 4 及更高版本中。

使用以下工作流，利用 source-to-image(S2I)流程在 OpenShift 镜像上为 JBoss EAP XP 构建并运行微服务应用。



注意

JBoss EAP XP 3.0.0 的 OpenShift 映像提供默认的独立配置文件，它基于 **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，请参见 *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 提供了一个小的特定工作示例，可用作您自己的项目的参考。如需更多信息，请参阅 JBoss EAP XP 3.0.0 随附的 **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 OpenShift 做好应用部署准备。

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 *在红帽客户门户网站中安装和配置 OpenShift Container Platform 集群*。

流程

1. 使用 **oc login 命令登录** 您的 OpenShift 实例。
2. 在 OpenShift 中创建新项目：
项目允许一组用户组织和管理与其他组不同的内容。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，使用以下命令创建名为 **eap-demo** 的新项目：

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

5.2. 配置 RED HAT CONTAINER REGISTRY 的身份验证

在为 JBoss EAP XP 导入和使用 OpenShift 镜像之前，您必须将身份验证配置到 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. 下载含有令牌 OpenShift 机密的 YAML 文件。
   您可以从令牌的 **Token Information** 页面上的 **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. 使用以下命令配置 OpenShift 项目的机密，并将以下机密名称替换为您在上一步中创建的机密名称：

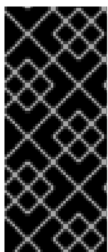
```
oc secrets link default 1234567-myserviceaccount-pull-secret --for=pull
oc secrets link builder 1234567-myserviceaccount-pull-secret --for=pull
```

其他资源

- 配置 Red Hat Container Registry 的身份验证
- 注册表服务帐户管理应用程序
- 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，它们将来可能会被删除。红帽会根据标准支持条款和条件继续提供完全支持和程序错误修复 OpenJDK 8 镜像和镜像流。

流程

1. 使用以下命令之一将 JBoss EAP XP 的 OpenShift 镜像的最新 JDK 11 镜像流和模板导入到您的 OpenShift 项目命名空间中。
   a. 导入 JDK 11 镜像流：

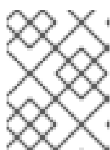
```
oc replace --force -f https://raw.githubusercontent.com/jboss-container-images/jboss-eap-openshift-templates/eap-xp3/jboss-eap-xp3-openjdk11-openshift.json
```

此命令导入以下镜像流和模板：

- JDK 11 构建器镜像流：jboss-eap-xp3-openjdk11-openshift
- JDK 11 运行时镜像流：jboss-eap-xp3-openjdk11-runtime-openshift

b. 导入 JDK 11 模板：

```
oc replace --force -f https://raw.githubusercontent.com/jboss-container-images/jboss-eap-openshift-templates/eap-xp3/templates/eap-xp3-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. 如果您有常规 **openshift** 命名空间的管理访问权限，并希望镜像流和模板可以被所有项目访问，请将 **-n openshift** 添加到 **命令的 oc replace** 行。例如：

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

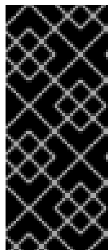
3. 如果要将镜像流和模板导入到其他项目中，请将 **-n *PROJECT_NAME*** 添加到 **命令的 oc replace** 行。例如：

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

如果使用 cluster-samples-operator，请参阅 OpenShift 文档中有关配置集群样本操作器的内容。有关配置集群示例操作器的详情，请查看 https://docs.openshift.com/container-platform/latest/openshift_images/configuring-samples-operator.html。

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

在 OpenShift 上部署 JBoss EAP XP 源至镜像(S2I)应用。



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，它们将来可能会被删除。红帽会根据标准的支持条款和条件继续为 OpenJDK 8 镜像和镜像流提供完全支持和程序错误修复。

先决条件

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

流程

1. 使用 JBoss EAP XP 镜像和您的 Java 应用的源代码，创建一个新的 OpenShift 应用。将其中一个提供的 JBoss EAP XP 模板用于 S2I 构建。

```
$ oc new-app --template=eap-xp3-basic-s2i \ (1)
 -p EAP_IMAGE_NAME=jboss-eap-xp3-openjdk11-openshift:latest \
 -p EAP_RUNTIME_IMAGE_NAME=jboss-eap-xp3-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-3.0.x \ (4)
 -p CONTEXT_DIR=microprofile-config (5)
```

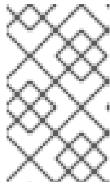
(1) 要使用的模板。应用镜像带有 **latest** 标签。

(2) 最新的镜像流和模板 已导入到项目命名空间中，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，您可能需要覆盖部分或全部默认值。要查看模板信息，包括参数列表和任何默认值，请使用命令 **oc describe template *TEMPLATE_NAME***。

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 Maven 构建的进度：

```
$ oc logs -f buildconfig/eap-xp3-basic-app-build-artifacts

…
Push successful
```

```
$ oc logs -f buildconfig/eap-xp3-basic-app
...
Push successful
```

其他资源

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

5.5. 为 JBOSS EAP XP SOURCE-TO-IMAGE(S2I)应用完成部署后的任务

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

部署后的任务示例包括：

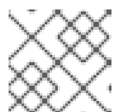
- 公开服务，以便可以从 OpenShift 外部查看应用。
- 将应用扩展为特定数量的副本。

流程

1. 使用以下命令获取应用的服务名称：

```
$ oc get service
```

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



注意

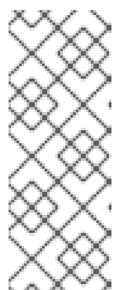
如果您使用模板来创建应用，则路由可能已存在。如果存在，请继续下一步。

```
$ oc expose service/eap-xp3-basic-app --port=8080
```

3. 获取路由的 URL。

```
$ oc get route
```

4. 使用 URL 访问 Web 浏览器中的应用。URL 是上一命令输出中 **HOST**/**PORT** 字段的值。



注意

对于 JBoss EAP XP 3.0.0 GA 分发，microprofile 配置快速入门不会回复对应用根上下文的 HTTPS GET 请求。这个增强只在 {JBossXPShortName101} GA 发行版中可用。

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

如果您的应用不使用 JBoss EAP 根上下文，请将应用的上下文附加到 URL。例如，对于 **microprofile-config** quickstart，URL 可能是 **http://*HOST_PORT_VALUE*/microprofile-config/**。

5. 您可以选择运行以下命令来扩展应用实例：此命令将副本数量增加到 3。

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

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

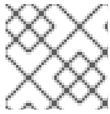
```
$ oc scale deploymentconfig/eap-xp3-basic-app --replicas=3
```

其它资源

如需有关 JBoss EAP XP 快速入门的更多信息，请参见《在 *JBoss EAP 中使用 MicroProfile* 指南》中的使用 快速入门章节。

第 6 章 功能修剪

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

仅 OpenShift 或构建可引导 JAR 时支持功能修剪。

其他资源

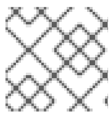
- 关于可引导 JAR

6.1. 可用的 JBOSS EAP 层

红帽提供了多个层，可用于自定义 OpenShift 中的 JBoss EAP 服务器调配或可引导 JAR。

三个层是提供核心功能的基础层。其他层是解耦器层，使用额外的功能增强基础层。

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。些层不支持 S2I 镜像；对层的描述请注意此限制。



注意

仅支持列出的层。不支持此处未列出的层。

6.1.1. 基础层

每个基础层都包含适用于典型服务器用户案例的核心功能。

datasources-web-server
此层包括一个 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 处理 1.1
- jakarta JSON Binding 1.0
- Jakarta Servlet 4.0
- Jakarta Expression Language 3.0
- Jakarta 服务器页面 2.3
- Jakarta Standard Tag Library 1.2
- jakarta Concurrency 1.1
- Jakarta 注释 1.3
- Jakarta XML Binding 2.3
- 雅加达调试支持其他语言 1.0

- Jakarta Transaction 1.3
- Jakarta Connector API 1.7

jaxrs-server
该层通过以下 JBoss EAP 子系统 **增强了数据源-web-server** 层：

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

此层还添加了基于 Infinispan 的第二级实体在容器中进行本地缓存。

此层中包含以下 MicroProfile 功能：

- MicroProfile REST 客户端

除了 **datasources-web-server** 层所支持的以下 Jakarta EE 规格外，还支持以下 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
该层使用以下 JBoss EAP 子系统增强了 **jaxrs-server** 层：

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

此层还会在 **jaxrs-server** 层中添加以下可观察功能：

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

除了 **jaxrs-server** 层支持的以下 Jakarta EE 规格外，还支持以下 Jakarta EE 规格：

- Jakarta 安全 1.0

6.1.2. decorator 层

解码器层不单独使用。您可以使用基础层配置一个或多个 decorator 层，以提供额外的功能。

ejb-lite

这个 decorator 层为调配的服务器添加了一个最小的 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 时才支持这一层。使用 S2I 时不支持这个层。

Jakarta Enterprise Beans
这个解码器层扩展了 **ejb-lite** 层。除了 **ejb-lite** 层中包含的基本功能外，这个层还会为置备的服务器增加以下支持：

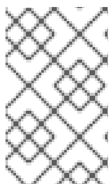
- MDB 实例池
- 远程连接器资源

如果要使用消息驱动型 Bean(MDB)或 Jakarta 企业 Bean 远程功能或两者，可使用此层。如果您不需要这些功能，请使用 **ejb-lite** 层。

只有在构建可引导 JAR 时才支持这一层。使用 S2I 时不支持这个层。

ejb-local-cache
此 decorator 层向调配的服务器中添加了对 Jakarta 企业 Bean 的本地缓存支持。

依赖项：您只能在包含 **ejb-lite** 层或 **ejb** 层时包括这个 层。



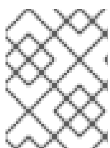
注意

这个层与 **ejb-dist-cache** 层不兼容。如果包含 **ejb-dist-cache** 层，则无法包含 **ejb-local-cache** 层。如果您同时包含这两个层，生成的构建可能包含意外的 Jakarta Enterprise Beans 配置。

只有在构建可引导 JAR 时才支持这一层。使用 S2I 时不支持这个层。

ejb-dist-cache
此 decorator 层向调配的服务器添加了对 Jakarta 企业 Bean 的分布式缓存支持。

依赖项：您只能在包含 **ejb-lite** 层或 **ejb** 层时包括这个 层。



注意

这个层与 **ejb-local-cache** 层不兼容。如果包含 **ejb-dist-cache** 层，则无法包含 **ejb-local-cache** 层。如果您同时包含这两个层，则生成的构建可能会导致意外的配置。

只有在构建可引导 JAR 时才支持这一层。使用 S2I 时不支持这个层。

jdr
此解码器层添加了 JBoss 诊断报告(**jdr**)子系统，以在请求红帽支持时收集诊断数据。

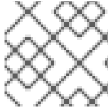
只有在构建可引导 JAR 时才支持这一层。使用 S2I 时不支持这个层。

Jakarta Persistence

这个解码器层为单节点服务器增加了持久性功能。请注意，只有服务器能够组成集群，分布式缓存才有效。

该层将 Hibernate 库添加到置备的服务器中，并提供以下支持：

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



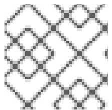
注意

这个层与 **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 时才支持这一层。使用 S2I 时不支持这个层。

jpa-distributed
此解码器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中，并提供以下支持：

- **jpa** 子系统的配置
- 配置 **infinispan** 子系统
- 本地 Hibernate 缓存容器
- 无效的和复制 Hibernate 缓存容器
- 配置 **jgroups** 子系统



注意

这个层与 **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 时才支持这一层。使用 S2I 时不支持这个层。

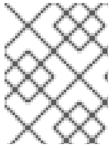
Jakarta Server Faces
此 decorator 层将 **jsf** 子系统添加到调配的服务器。

只有在构建可引导 JAR 时才支持这一层。使用 S2I 时不支持这个层。

microprofile-platform
此 decorator 层将以下 MicroProfile 功能添加到调配的服务器中：

- MicroProfile Config
- MicroProfile 容错
- MicroProfile Health
- MicroProfile JWT
- MicroProfile Metrics
- MicroProfile OpenAPI

- MicroProfile OpenTracing

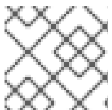


注意

此层包含也会包含在 **可观察** 层中的 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以下可观察功能：

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



注意

此层内置到 **cloud-server** 层。您不需要将此层添加到 **云服务器**层。

remote-activemq
此 decorator 层添加了与远程 ActiveMQ 代理与调配的服务器通信的功能，从而集成消息传递支持。

池式连接工厂配置将 **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 时才支持这一层。使用 S2I 时不支持这个层。

sso
这个解码器层将红帽单点登录集成添加到调配的服务器中。

只有在使用 S2I 调配服务器时，才应使用此层。

web-console
此 decorator 层将管理控制台添加到调配的服务器。

只有在构建可引导 JAR 时才支持这一层。使用 S2I 时不支持这个层。

web-clustering
此decorator 层通过配置基于非本地 Infinispan 的容器 web 缓存来增加对可分布式 Web 应用的支持，以便处理适合集群环境的数据会话。

web-passivation
此后方层通过为适合单一节点环境的数据传输配置基于 Infinispan 的容器 Web 缓存来配置对可分发的 Web 应用的支持。

只有在构建可引导 JAR 时才支持这一层。使用 S2I 时不支持这个层。

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

只有在构建可引导 JAR 时才支持这一层。使用 S2I 时不支持这个层。

其他资源

- 池连接事实属性

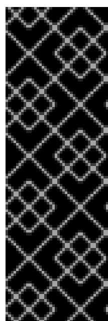
第 7 章 在红帽 CODEREADY STUDIO 上为 JBOSS EAP 启用 MICROPROFILE 应用开发

如果要将 MicroProfile 功能整合到在 CodeReady Studio 上开发的应用中，您必须在 CodeReady Studio 中启用对 JBoss EAP 的 MicroProfile 支持。

JBoss EAP 扩展包为 MicroProfile 提供支持。

JBoss EAP 扩展软件包在 JBoss EAP 7.2 及更早版本中不受支持。

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 扩展包支持和生命周期政策页。



重要

Openshift 的 JBoss EAP XP 快速入门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。

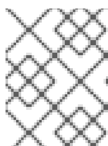
如需有关 技术预览功能支持范围 的信息，请参阅红帽客户门户网站中的技术预览功能支持范围。

7.1. 配置 CODEREADY STUDIO 以使用 MICROPROFILE 功能

要在 JBoss EAP 上启用 MicroProfile 支持，请注册 JBoss EAP XP 的新运行时服务器，然后创建新的 JBoss EAP 7.4 服务器。

为服务器命名合适的名称，以帮助您识别它支持 MicroProfile 功能。

此服务器使用新创建的 JBoss EAP XP 运行时，它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

- JBoss EAP XP 3.0.0 已安装。

流程

1. 在 **New Server** 对话框中设置新服务器。
   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 **Next**。

2. 配置新服务器.

   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-7.3*。

   b. 确保 **执行环境** 设置为 *JavaSE-1.8*。

   c. 可选：更改 **服务器基础目录和 配置文件中的** 值。

   d. 点 **Finish**。

结果

您现在已准备好使用 MicroProfile 功能开始开发应用，或者开始将 MicroProfile 快速入门用于 JBoss EAP。

7.2. 将 MICROPROFILE 快速入门用于 CODEREADY STUDIO

启用 MicroProfile 快速入门提供了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 容错
- MicroProfile Health
- MicroProfile JWT
- MicroProfile Metrics
- MicroProfile OpenAPI
- MicroProfile OpenTracing
- MicroProfile REST 客户端

流程

1. 从 Quickstart Parent Artifact 中导入 **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** 对话框的启动配置上定义环境变量。

   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 企业应用平台扩展包

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 插件将微服务应用构建和打包为可引导 JAR。然后，您可以在 JBoss EAP 裸机平台或 JBoss EAP OpenShift 平台上运行该应用。

8.1. 关于可引导 JAR

您可以使用 JBoss EAP JAR Maven 插件将微服务应用构建和打包为可引导 JAR。

可引导 JAR 包含服务器、打包的应用和启动服务器的运行时。

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空间。因此，您可以根据要求配置服务器，包括仅提供所需功能的 Galleon 层。

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

可引导 JAR 类似于标准的 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。

将服务器打包到可引导 JAR 中时存在以下限制：

- 不支持需要重新启动服务器的 CLI 管理操作。
- 服务器无法在仅 admin 模式下重新启动，这是一种启动与服务器管理相关的服务的模式。
- 如果关闭服务器，应用到服务器的更新将会丢失。

另外，您可以调配可引导 JAR。此 JAR 仅包含服务器，因此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 插件将应用构建为可引导 JAR。

您可以从 Maven 存储库检索最新的 Maven 插件版本，该存储库可通过 /ga/org/wildfly/plugins/wildfly-jar-maven-plugins/wildfly-jar-maven-plugin 提供。

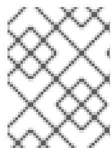
在 Maven 项目中，src **目录包含** 构建应用所需的所有源文件。在 JBoss EAP JAR Maven 插件构建可引导 JAR 后，生成的 JAR 位于 **target/<application>-bootable.jar** 中。

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 功能包及其部分层来自定义服务器配置文件。
- 支持向打包的可引导 JAR 中添加额外的文件，如密钥存储文件。
- 包括创建可引导 JAR 的 hollow JAR 的功能；即，可引导 JAR，不包含应用。

在使用 JBoss EAP JAR Maven 插件创建可引导 JAR 后，您可以通过发出下列命令来启动应用：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

要获得受支持的可引导 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。
- 有关为您的项目构建功能包支持的 Galleon 插件的详情，请查看 WildFly Galleon Maven 插件文档。
- 如需有关选择配置 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，以了解与可引导 JAR 一起使用的参数。

表 8.1. 支持的可引导 JAR 可执行参数

参数	Description
--help	显示指定命令的帮助消息并退出。
--deployment=<path>	特定于可引导 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	打印生成的 Galleon 配置文件的内容。
--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** 参数来指定要安装服务器的目录。
-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，b **management=IP_ADDRESS** 。

参数	Description
-b=<value>	设置系统属性 **jboss.bind.address**，用于为公共接口配置绑定地址。如果未指定值，则默认为 127.0.0.1。
-D<name>[=<value>]	指定服务器在服务器运行时设置的系统属性。可引导 JAR JVM 不设置这些系统属性。
--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. 为您的可引导 JAR 服务器指定 GALLEON 层

您可以指定 Galleon 层来为您的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。

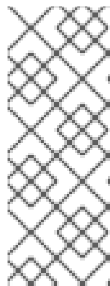
要引用单个功能包，请使用 **<feature-pack-location>** 元素来指定其位置。以下示例在 Maven 插件配置文件中的 **<feature -pack>** 元素中指定了 **org.jboss.eap:wildfly-galleon-pack: 3.0.0.GA-redhat-00001**。

```
<configuration>
  <feature-pack-location>org.jboss.eap:wildfly-galleon-pack:3.0.0.GA-redhat-00001</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 功能包添加到 **<feature-packs>** 元素中：

```
<configuration>
    <feature-packs>
        <feature-pack>
            <location>org.jboss.eap:wildfly-galleon-pack:3.0.0.GA-redhat-00001</location>
        </feature-pack>
        <feature-pack>
            <location>org.jboss.sso:keycloak-adapter-galleon-pack:9.0.10.redhat-00001</location>
        </feature-pack>
    </feature-packs>
</configuration>
```

您可以组合多个功能包中的 Galleon 层来配置可引导 JAR 服务器，使其仅包含提供您所需功能的受支持 Galleon 层。



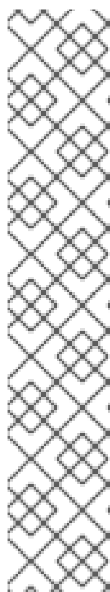
注意

在裸机平台上，如果您未在配置文件中指定 Galleon 层，调配的服务器包含与默认 **standalone-microprofile.xml** 配置相同的配置。

在 OpenShift 平台上，在插件 **配置中添加 <cloud/>** 配置元素后，您选择不在配置文件中指定 Galleon 层，置备的服务器包含根据云环境进行调整的配置，并且与默认的 **standalone-microprofile-ha.xml** 类似。

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 功能包版本，如 **3.0.X.GA-redhat-*BUILD_NUMBER***，其中 *X* 是 JBoss EAP XP 和 *BUILD_NUMBER* 的微版本，是 Galleon 功能包的构建号。*X* 和 *BUILD_NUMBER* 在 JBoss EAP XP 3.0.0 产品生命周期中都可能会演变。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：

- 用于 Maven 插件版本的 **${bootable.jar.maven.plugin.version}**。
- Gall **eon 功能包版本的 ${JBoss.xp.galleon.feature.pack.version}**。

您必须在项目中设置这些属性。例如：

```
<properties>
   <bootable.jar.maven.plugin.version>4.0.3.Final-redhat-
00001</bootable.jar.maven.plugin.version>
   <jboss.xp.galleon.feature.pack.version>3.0.0.GA-redhat-
00001</jboss.xp.galleon.feature.pack.version>
</properties>
```

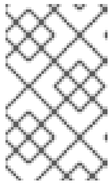
流程

1. 确定受支持的 JBoss EAP Galleon 层，它们可提供运行应用所需的功能。
2. 在 Maven 项目 **pom.xml** 文件的 **<plugin>** 元素中引用 JBoss EAP 功能包位置。您必须指定任何 Maven 插件的最新版本，以及 **org.jboss.eap:wildfly-galleon-pack** Galleon 功能包的最新版本，如下例中所示。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-distributed** 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>
```

此示例还演示了将 **jpa** 层从项目中排除。



注意

如果您在项目中包含 **jpa-distributed** 层，则必须将 **jpa** 层从 **jaxrs-server** 层中排除。**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

- 有关可用基础层的详情，请参考 基本层。
- 有关为您的项目构建功能包支持的 Galleon 插件的详情，请查看 WildFly Galleon Maven 插件文档。
- 有关选择用于配置 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 Maven 依赖项的信息，请参阅 *Apache Maven Project* 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。

可引导 JAR 包含服务器、打包的应用和启动服务器的运行时。

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

在构建必须在可引导 JAR 中打包的 web 应用程序时，您必须在 **pom.xml** 文件的 **<packaging>** 元素 中指定 **war**。例如：

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

构建应用需要此值，才能将构建应用打包为 WAR 文件，而不是默认的 JAR 文件。

在仅用于构建易引导 JAR 的 Maven 项目中，将打包值设置为 **pom**。例如：

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

在为 Maven 项目构建可引导 JAR 时，您不限于使用 **pom** 打包。您可以通过在 **<hollow-jar>** 元素中为任意类型的打包（如 **war** ）指定 **true** 来创建一个。请参阅在 JBoss EAP 裸机平台上创建可引导 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 功能包版本，如 **3.0.X.GA-redhat-*BUILD_NUMBER***，其中 *X* 是 JBoss EAP XP 和 *BUILD_NUMBER* 的微版本，是 Galleon 功能包的构建号。*X* 和 *BUILD_NUMBER* 在 JBoss EAP XP 3.0.0 产品生命周期中都可能会演变。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，并且添加了用于创建 MicroProfile 应用的依赖关系。请参阅 MicroProfile 配置开发。

注意

流程中显示的示例指定以下属性：

- 用于 Maven 插件版本的 **${bootable.jar.maven.plugin.version}**。
- Gall **eon 功能包版本的 ${JBoss.xp.galleon.feature.pack.version}**。

您必须在项目中设置这些属性。例如：

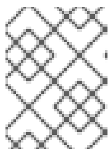
```
<properties>
   <bootable.jar.maven.plugin.version>4.0.3.Final-redhat-
00001</bootable.jar.maven.plugin.version>
   <jboss.xp.galleon.feature.pack.version>3.0.0.GA-redhat-
00001</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

1. 将以下内容添加到 **pom.xml** 文件的 **<build>** 元素 中。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

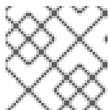
如果您没有在 **pom.xml** 文件中指定 Galleon 层，则可引导 JAR 服务器包含与 **standalone-microprofile.xml** 配置一致的配置。

2. 将应用程序打包为可引导 JAR:

```
$ mvn package
```

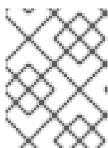
3. 启动应用程序：

```
$ 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

要查看受支持的可引导 JAR 参数列表，请在 **java -jar target/microprofile-config-bootable.jar** 命令的末尾附加 **--help**。

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

- 如需有关可用 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 HOLLOW 可引导 JAR

您可以将应用打包为 JBoss EAP 裸机平台上的可热引导 JAR。

易引导 JAR 仅包含 JBoss EAP 服务器。hollow 可引导 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，Hollow 可引导 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的依赖项。请参阅 MicroProfile 配置开发。
- 您已完成了在 JBoss EAP 裸机平台上使用可引导 JAR 中所述的 **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 功能包版本，如 **3.0.X.GA-redhat-*BUILD_NUMBER***，其中 *X* 是 JBoss EAP XP 和 *BUILD_NUMBER* 的微版本，是 Galleon 功能包的构建号。*X* 和 *BUILD_NUMBER* 在 JBoss EAP XP 3.0.0 产品生命周期中都可能会演变。请参阅 /ga/org/jboss/eap/wildfly-galleon-pack 的索引。

注意

流程中演示的示例为 Gall **eon 功能包版本指定了 ${jboss.xp.galleon.feature.pack.version}**，但您必须在项目中设置 属性。例如：

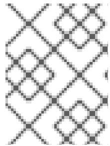
```
<properties>
    <jboss.xp.galleon.feature.pack.version>3.0.0.GA-redhat-
00001</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

1. 要构建 hollow 可引导 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

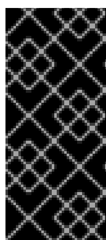
通过在 **<hollow-jar>** 元素中指定 **true**，JBoss EAP JAR Maven 插件不会将应用程序包含在 JAR 中。

1. 构建 hollow 可引导 JAR:

```
$ mvn clean package
```

2. 运行 hollow 可引导 JAR:

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



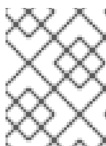
重要

要指定您要部署到服务器上的 WAR 文件的路径，请使用以下参数，其中 **<PATH_NAME>** 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

若要将 Web 应用注册到根目录，请将 applicationROOT.war 命名为 application **ROOT.war**。

其他资源

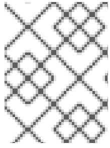
- 如需有关可用 MicroProfile 功能的信息，请参阅 MicroProfile 配置。
- 有关 JBoss EAP XP 3.0.0 支持的 JBoss EAP JAR Maven 插件的更多信息，请参阅 JBoss EAP Maven 插件。

8.7. CLI 脚本

您可以在可引导 JAR 打包期间创建 CLI 脚本来配置服务器。

CLI 脚本是一个文本文件，它包含一系列 CLI 命令，可用于应用其他服务器配置。例如，您可以创建一个脚本来添加新日志记录器到 **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一个命令中，为管理 HTTP 端点启用 HTTP 身份验证。



注意

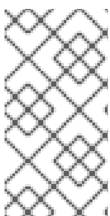
在将应用程序打包为可引导 JAR 之前，您必须在插件配置的 **<cli-session> 元素** 中定义 CLI 脚本。这可确保服务器配置设置在打包可引导 JAR 后保留。

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，但限制确实存在。例如，在打包可引导 JAR 时，您不能使用 Galleon 层启用 HTTPS **undertow** 侦听器。相反，您必须使用 CLI 脚本。

您必须在 **pom.xml** 文件的 **<cli-session>** 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	Description
script-files	到脚本文件的路径列表.
properties-file	指定属性文件路径的可选属性。此文件列出了脚本可使用 **${my.prop}** 语法引用的 Java 属性。以下示例将 **public inet-address** 设置为 **all.addresses**:**/interface=public:write-attribute(name=inet-address,value=${all.addresses})**
resolve-expressions	包含布尔值的可选属性。指示是否在向服务器发送操作请求前解析系统属性或表达式。默认值为 **true**。

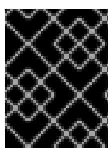


注意

- CLI 脚本按照在 **pom.xml** 文件的 **<cli-session>** 元素中定义的顺序启动。
- 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的服务器。

8.8. 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 平台上运行该应用。



重要

在 OpenShift 中，您不能将 EAP Operator 自动事务恢复功能用于可引导 JAR。计划在以后的 JBoss EAP XP 3.0.0 补丁版本中修复此技术限制。

先决条件

- 您已为 MicroProfile Config 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 功能包版本，如 **3.0.X.GA-redhat-*BUILD_NUMBER***，其中 *X* 是 JBoss EAP XP 3 和 *BUILD_NUMBER* 的微版本，是 Galleon 功能包的构建号。*X* 和 *BUILD_NUMBER* 在 JBoss EAP XP 3.0.0 产品生命周期中都可能会演变。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：

- 用于 Maven 插件版本的 **${bootable.jar.maven.plugin.version}**。
- Gall **eon 功能包版本的 ${JBoss.xp.galleon.feature.pack.version}**。

您必须在项目中设置这些属性。例如：

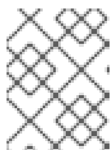
```
<properties>
    <bootable.jar.maven.plugin.version>4.0.3.Final-redhat-
00001</bootable.jar.maven.plugin.version>
    <jboss.xp.galleon.feature.pack.version>3.0.0.GA-redhat-
00001</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

1. 将以下内容添加到 **pom.xml** 文件的 **<build>** 元素 中。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

您必须将 **<cloud/>** 元素包含在插件配置的 **<configuration> 元素** 中，以便 JBoss EAP Maven JAR 插件可以识别您选择 OpenShift 平台。

2. 打包应用程序：

```
$ mvn package
```

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

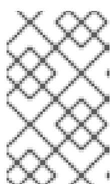
4. 在 OpenShift 中创建新项目：例如：

```
$ 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. 使用 **target/openshift** 子目录作为二进制输入来构建应用程序。



注意

OpenShift 应用一组 CLI 脚本命令到可引导 JAR 配置文件，以将其调整到云环境。您可以通过打开 Maven 项目 **/target 目录中的 bootable-jar-build-artifacts/generated-cli-script.txt** 文件来访问此脚本。

6. *验证：*
   使用以下命令，查看可用的 OpenShift pod 列表并检查 pod 构建状态：

```
$ oc get pods
```

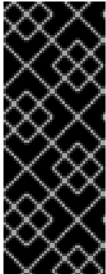
验证构建的应用程序镜像：

```
$ 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会显示 **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 **JAVA_OPTS_APPEND** 环境变量的用法：

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

创建一个新应用并启动。应用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

- 如需有关 MicroProfile 的信息，请参阅 MicroProfile 配置。
- 有关 **ConfigSources** 的信息，请参阅 默认 MicroProfile 配置属性。

8.9. 为 OPENSHIFT 配置可引导 JAR

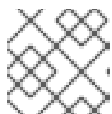
在使用可引导 JAR 之前，您可以配置 JVM 设置以确保您的单机服务器在用于 OpenShift 的 JBoss EAP 上正确运行。

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来设置属性的值。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信息。



注意

如果需要为可引导 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 关联的应用程序，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 的 OpenShift S2I 镜像的 JVM，但会有以下区别：

- 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 metaspace 大小，您可以设置 **GC_METASPACE_SIZE** 环境变量。要获得最佳元数据容量性能，请将值设置为 **96**。
- **GC_MAX_METASPACE_SIZE** 的默认值被设置为 **100**，但对于垃圾回收后的最佳元数据容量，您必须将其设置为至少 **256**。
- 为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**。

在导入的 OpenJDK 11 镜像上运行时，这些配置提高了 JVM 的内存设置和垃圾回收功能。

8.10. 在 OPENSHIFT 上的应用程序中使用 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容器集中。

ConfigMap 是一种 OpenShift 资源，用于在键值对中存储非机密数据。

在指定了 **microprofile-platform** Galleon 层以 add **microprofile-config-smallrye** 子系统和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 ConfigSource**。您可以在 Maven 项目的根目录下将 CLI 脚本保存在可访问的目录中，如 **/scripts** 目录。

MicroProfile 配置功能在 JBoss EAP 中使用 SmallRye Config 组件实施，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

- 已安装 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 平台上运行该应用。有关在 OpenShift 平台上将应用构建为可引导 JAR 的信息，请参阅 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 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})
```

The **mp-config.cli** CLI 脚本创建一个新的 **ConfigSource**，其从属性文件检索到序数和路径值。

4. 将脚本保存到 **/scripts** 目录中，该目录位于项目的根目录下。

5. 将以下配置提取添加到现有插件 **<configuration>** 元素中：

```
<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** 您的 OpenShift 实例。

8. *可选：* 如果您之前没有创建 **target/openshift** 子目录，则必须使用以下命令创建 suddirectory：

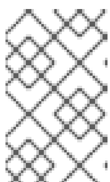
```
$ mkdir target/openshift
```

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，以将其用于云环境。您可以通过打开 Maven 项目 **/target** 目录中的 **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** 挂载到应用中。例如：

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

在执行 **oc set volume** 命令后，应用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

- 如需有关 MicroProfile Config **ConfigSources** 属性的信息，请参阅 默认 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。

8.11. 创建可引导 JAR MAVEN 项目

按照以下步骤创建 Maven 项目示例。您必须先创建一个 Maven 项目，然后才能执行以下步骤：

- 为可引导 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 JBoss EAP 可引导 JAR 应用程序

在 **pom.xml** 项目文件中，您可以将 Maven 配置为检索构建可引导 JAR 所需的项目构件。

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 构件的版本，请将 BOM 添加到 **pom.xml** 项目的 **<dependencyManagement> 部分中**。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 构件添加到项目的 **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 插件的信息，请参阅 JBoss EAP Maven 插件。
- 有关 Galleon 层的详情，请参考指定可引导 JAR 服务器的 Galleon 层。
- 有关将红帽单点登录 Galleon 功能包包含在项目中的更多信息，请参阅使用红帽单点登录保护 JBoss EAP 可引导 JAR 应用程序。

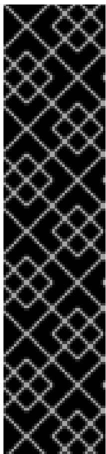
8.12. 为可引导 JAR 启用 JSON 日志记录

您可以通过使用 CLI 脚本配置服务器日志配置，为您的可引导 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 JSON 格式的日志消息。

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

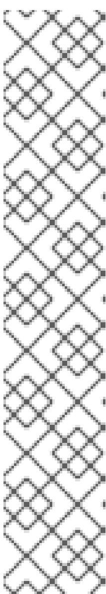
- 您已检查了最新的 Maven 插件版本，如 ***MAVEN_PLUGIN_VERSION*.X.GA.Final-redhat-00001**，其中 *MAVEN_PLUGIN_VERSION* 是主版本，*X* 是 microversion。请参阅 /ga/org/wildfly/plugins/wildfly-jar-maven-plugin 的索引。
- 您已检查了最新的 Galleon 功能包版本，如 **3.0.X.GA-redhat-*BUILD_NUMBER***，其中 *X* 是 JBoss EAP XP 和 *BUILD_NUMBER 的* 次要版本，是 Galleon 功能包的构建号。*X* 和 *BUILD_NUMBER* 在 JBoss EAP XP 3.0.0 产品生命周期中都可能会演变。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：

- 用于 Maven 插件版本的 **${bootable.jar.maven.plugin.version}**。
- Gall **eon 功能包版本的 ${JBoss.xp.galleon.feature.pack.version}**。

您必须在项目中设置这些属性。例如：

```
<properties>
    <bootable.jar.maven.plugin.version>4.0.3.Final-redhat-00001</bootable.jar.maven.plugin.version>
    <jboss.xp.galleon.feature.pack.version>3.0.0.GA-redhat-00001</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

1. 将由 BOM 管理的 JBoss Logging 和 Jakarta RESTful Web Services 依赖项添加到项目的 **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. 将以下内容添加到 **pom.xml** 文件的 **<build> 元素** 中。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** 是含有应用的 **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 CLI 脚本，如 **logging.cli**，并将它保存在可引导 JAR 中的可访问目录中，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. 将以下配置提取添加到插件 **<configuration>** 元素中：

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

本例显示了 **logging.cli** CLI 脚本，该脚本修改服务器日志配置文件，以为您的应用启用 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，请按照在 JBoss EAP 裸机平台上使用可引导 JAR 中所述的步骤进行操作， 但会有以下差异：

   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 ：
   预期输出：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** 调试追踪。

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. 重新构建应用程序：

    ```
    $ mvn clean package
    ```

    c. 使用 **oc login 命令登录** 您的 OpenShift 实例。

    d. 在 OpenShift 中创建新项目：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：*运行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
```

访问应用的正在运行的容器集日志。其中 **APP_POD_NAME** 是正在运行的容器集日志记录应用程序的名称。

```
$ oc logs APP_POD_NAME
```

预期结果：pod 日志采用 JSON 格式，包括 **com.example.logging.HelloWorldEndpoint** 调试追踪。

其他资源

- 有关 JBoss EAP 日志功能的信息，请参阅《*配置指南*》 中的使用 JBoss EAP 进行日志记录。
- 有关在 OpenShift 中使用可引导 JAR 的详情，请参考 在 JBoss EAP OpenShift 平台上使用可引导 JAR。
- 有关为您的项目指定 JBoss EAP JAR Maven 的详情，请参考指定可引导 JAR 服务器的 Galleon 层。
- 有关创建 CLI 脚本的详情，请参考 CLI 脚本。

8.13. 为多个可引导 JAR 实例启用 WEB 会话数据存储

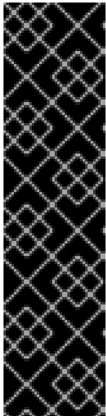
您可以将 web 集群应用程序构建和打包为可引导 JAR。

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 功能包版本，如 **3.0.X.GA-redhat-*BUILD_NUMBER***，其中 *X* 是 JBoss EAP XP 和 *BUILD_NUMBER* 的微版本，是 Galleon 功能包的构建号。*X* 和 *BUILD_NUMBER* 在 JBoss EAP XP 3.0.0 产品生命周期中都可能会演变。请参阅

/ga/org/jboss/eap/wildfly-galleon-pack 的索引。

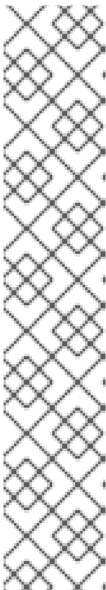
- 您已创建了 Maven 项目，设置父依赖项，并添加了用于创建 web-clustering 应用的依赖项。请参阅 创建可引导 JAR Maven 项目 。



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：

- 用于 Maven 插件版本的 **${bootable.jar.maven.plugin.version}**。
- Gall **eon 功能包版本的 ${JBoss.xp.galleon.feature.pack.version}**。

您必须在项目中设置这些属性。例如：

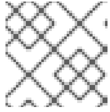
```
<properties>
    <bootable.jar.maven.plugin.version>4.0.3.Final-redhat-
00001</bootable.jar.maven.plugin.version>
    <jboss.xp.galleon.feature.pack.version>3.0.0.GA-redhat-
00001</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

1. 将以下内容添加到 **pom.xml** 文件的 **<build> 元素** 中。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

本例使用 **web-clustering** Galleon 层来启用 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>
```

<distributable/> 标签表示此 servlet 可以分布在多个服务器中。

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** 是含有应用的 **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，请按照在 JBoss EAP 裸机平台上使用可引导 JAR 中所述的步骤进行操作，但会有以下差异：

   a. 在 JBoss EAP 裸机平台上，您可以使用 **java -jar** 命令来运行多个可引导 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 实例上的应用 ：注意用户会话 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，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. 重新构建应用程序：

```
$ mvn clean package
```

c. 使用 **oc login 命令登录** 您的 OpenShift 实例。

d. 在 OpenShift 中创建新项目：例如：

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

e. 若要在 JBoss EAP OpenShift 平台上运行 web-clustering 应用，必须为容器集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。以下示例演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
```

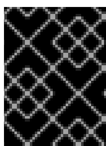
f. 输入以下 **oc** 命令来创建应用程序镜像：

```
$ 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 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** 环境变量来查看当前 OpenShift 命名空间中的其他 pod；否则，服务器会尝试从 **默认** 命名空间中检索 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
```

请注意用户会话 ID 和用户创建时间。

j. 将应用程序扩展为两个 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* 是您最旧 pod 的名称。

m. 再次访问应用程序：

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

预期结果：新容器集生成的会话 ID 和创建时间与已终止 pod 的 相匹配。这表示启用了 Web 会话数据存储。

其他资源

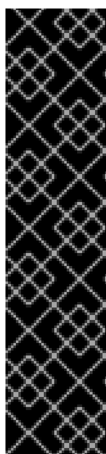
- 有关可分布式 Web 会话管理配置文件的信息，请参阅《*开发指南*》 中适用于分布式 Web 会话配置的 distribut-web 子系统。
- 有关配置 JGroups 协议堆栈的信息，*请参阅 OpenShift Container Platform 入门指南中的配置* JGroups 发现机制。

8.14. 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。此脚本将安全域和安全域添加到您的服务器。

先决条件

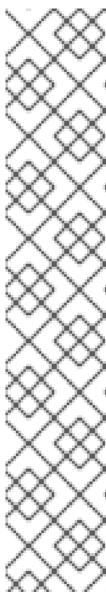
- 您已检查了最新的 Maven 插件版本，如 ***MAVEN_PLUGIN_VERSION*.X.GA.Final-redhat-00001**，其中 *MAVEN_PLUGIN_VERSION* 是主版本，*X* 是 microversion。请参阅 /ga/org/wildfly/plugins/wildfly-jar-maven-plugin 的索引。
- 您已检查了最新的 Galleon 功能包版本，如 **3.0.X.GA-redhat-*BUILD_NUMBER***，其中 *X* 是 JBoss EAP XP 和 *BUILD_NUMBER* 的微版本，是 Galleon 功能包的构建号。*X* 和 *BUILD_NUMBER* 在 JBoss EAP XP 3.0.0 产品生命周期中都可能会演变。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，并添加了用于创建需要 HTTP 身份验证的应用的依赖关系。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：

- 用于 Maven 插件版本的 **${bootable.jar.maven.plugin.version}**。
- Gall **eon 功能包版本的 ${JBoss.xp.galleon.feature.pack.version}**。

您必须在项目中设置这些属性。例如：

```
<properties>
    <bootable.jar.maven.plugin.version>4.0.3.Final-redhat-00001</bootable.jar.maven.plugin.version>
    <jboss.xp.galleon.feature.pack.version>3.0.0.GA-redhat-00001</jboss.xp.galleon.feature.pack.version>
</properties>
```

流程

1. 将以下内容添加到 **pom.xml** 文件的 **<build>** 元素 中。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>
```

示例显示了包含 **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 CLI 脚本，如 **authentication.cli**，并将它保存在可引导 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. 将以下配置提取添加到插件 **<configuration>** 元素中：

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

本例显示了 **authentication.cli** CLI 脚本，该脚本会将默认 **undertow** 安全域配置为为服务器定义的安全域。

7. 在 Maven 项目的根目录中创建一个目录，以存储 JBoss EAP JAR Maven 插件添加到可引导 JAR 的属性文件：

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

其中 **APPLICATION_ROOT** 是含有应用的 **pom.xml** 配置文件的目录。

此目录存储 **可引导用户.properties 和可 引导组.properties** 文件等文件。

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** 元素添加到现有的 **<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
```

11. 调用 servlet，但不要指定凭证：

```
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据。例如：

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

返回 HTTP 200 状态，表示已为您的可引导 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的 CLI Security 命令为应用程序*启用 HTTP 身份验证。

8.15. 使用红帽单点登录保护 JBOSS EAP 可引导 JAR 应用程序

您可以使用 Galleon **keycloak-client-oidc** 层来安装由 Red Hat Single Sign-On 7.4 OpenID Connect 客户端适配器置备的服务器版本。

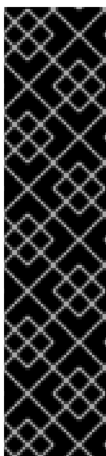
keycloak-client-oidc 层为 Maven 项目提供红帽单点登录 OpenID Connect 客户端适配器。这个层包含在 Red Hat Single Sign **-On 功能包的 keycloak-adapter-galleon-pack** 中。

您可以将 **keycloak-adapter-galleleon-pack** 功能包添加到 JBoss EAP Maven 插件配置中，然后添加 **keycloak-client-oidc**。您可以通过访问支持 的配置来查看与 JBoss EAP 兼容的红帽单点登录客户端适配器：红帽单点登录 7.4 网页。

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 可引导 JAR。

先决条件

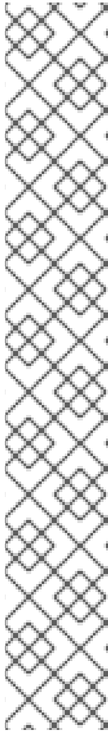
- 您已检查了最新的 Maven 插件版本，如 ***MAVEN_PLUGIN_VERSION*.X.GA.Final-redhat-00001**，其中 *MAVEN_PLUGIN_VERSION* 是主版本，*X* 是 microversion。请参阅 /ga/org/wildfly/plugins/wildfly-jar-maven-plugin 的索引。
- 您已检查了最新的 Galleon 功能包版本，如 **3.0.X.GA-redhat-*BUILD_NUMBER***，其中 *X* 是 JBoss EAP XP 和 *BUILD_NUMBER* 的微版本，是 Galleon 功能包的构建号。*X* 和 *BUILD_NUMBER* 在 JBoss EAP XP 3.0.0 产品生命周期中都可能会演变。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 功能包版本，如 **org.jboss.sso:keycloak-adapter-galleon-pack:9.0.X:redhat-BUILD_NUMBER**，其中 **X** 是红帽单点登录的微版本，它依赖于用于保护应用程序的红帽单点登录服务器版本，和 **BUILD_NUMBER** 是红帽单点登录 Galleon 功能包的构建号。*X* 和 *BUILD_NUMBER* 在 JBoss EAP XP 3.0.0 产品生命周期中都可能会演变。请参阅 /ga/org/sso/keycloak-adapter-galleon-pack 索引.
- 您已创建了 Maven 项目，设置父依赖项，并添加了依赖项，以创建您希望通过红帽单点登录保护的应用程序。请参阅 创建可引导 JAR Maven 项目。
- 您有一个在端口 8090 上运行的红帽单点登录服务器。请参阅 启动红帽单点登录服务器。
- 您已登陆到 Red Hat Single Sign-On 管理控制台并创建了以下元数据：
  - 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。流程中的示例将 **simple-webapp** 定义为 Web 应用，**http://localhost:8080/simple-webapp/secured** 定义为根 URL。



重要

在设置 Maven 项目时，您必须在 Maven archetype 中为您要通过红帽单点登录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：

- 用于 Maven 插件版本的 **${bootable.jar.maven.plugin.version}**。
- Gall **eon 功能包版本的 ${JBoss.xp.galleon.feature.pack.version}**。
- 用于红帽单点登录功能包版本的 **${Keycloak.feature.pack.version}**。

您必须在项目中设置这些属性。例如：

```
<properties>
    <bootable.jar.maven.plugin.version>4.0.3.Final-redhat-00001</bootable.jar.maven.plugin.version>
    <jboss.xp.galleon.feature.pack.version>3.0.0.GA-redhat-00001</jboss.xp.galleon.feature.pack.version>
    <keycloak.feature.pack.version>9.0.10.redhat-00001</keycloak.feature.pack.version>
</properties>
```

流程

1. 将以下内容添加到 **pom.xml** 文件的 **<build> 元素** 中。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.jboss.sso: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 层通过使用 **keycloak** 子系统及其依赖项激活对红帽单点登录身份验证的支持，为项目提供红帽单点 OpenID Connect 客户端适配器。红帽单点登录客户端适配器是使用红帽单点登录保护应用程序和服务库。

2. 在项目的 **pom.xml** 文件中，在插件配置中将 **<context-root>** 设置为 **false**。这会将应用注册到 **simple-webapp** 资源路径。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 CLI 脚本，如 **configure-oidc.cli**，并将它保存在可引导 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。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="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   metadata-complete="false">

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

```
    </login-config>

</web-app>
```

6. *可选：* 除了第 7 到第 9 步外，您还可以通过将 **keycloak.json** 描述符添加到 web 应用的 **WEB-INF** 目录，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，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，使其包含以下内容，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。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. 在网页浏览器中指定以下 URL 以访问通过红帽单点登录保护的网页。以下示例显示了受保护的 **simple-webapp** Web 应用程序的 URL：

```
http://localhost:8080/simple-webapp/secured
```

11. 以红帽单点登录域中的用户身份登录。

12. *验证：* 检查网页是否显示以下输出：

```
Current Principal '<principal id>'
```

其他资源

- 有关配置红帽单点登录适配器子系统的详情，请参考《*保护应用和服务指南*》*中的 JBoss EAP 适配器*。
- 有关为您的项目指定 JBoss EAP JAR Maven 的详情，请参考指定可引导 JAR 服务器的 Galleon 层。

8.16. 在 DEV 模式下打包可引导 JAR

JBoss EAP JAR Maven 插件开发 **目标** 提供 **开发模式开发** 模式，可用于增强应用程序开发流程。

在 **dev** 模式中，在更改应用程序后不需要重建可引导 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，并且添加了用于创建 MicroProfile 应用的依赖关系。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 **目标/部署目录**。

2. 提示 JBoss EAP Maven 插件使用以下命令构建应用程序并将其复制到 **目标/部署目录中** ：

   ```
   $ mvn package -Ddev
   ```

   打包在可引导 JAR 内的服务器将部署存储在 **目标/部署目录中的** 应用。

3. 修改应用程序代码中的代码。

4. 使用 **mvn package -Ddev** 提示 JBoss EAP Maven 插件重新构建应用并重新部署。

5. 停止服务器。例如：

   ```
   $ mvn wildfly-jar:shutdown
   ```

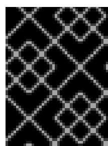
6. 完成应用程序更改后，将应用程序打包为可引导 JAR：

   ```
   $ mvn package
   ```

8.17. 将 JBOSS EAP 补丁应用到可引导 JAR

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。

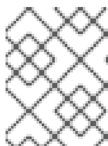
CLI 脚本发出 **patch apply** 命令，以在可引导 JAR 构建期间应用补丁。



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。您必须在没有补丁的情况下重建可引导 JAR。

此外，您还可以利用 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>** 配置选项来引用用于修补服务器的 CLI 脚本。



注意

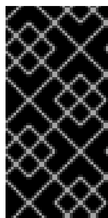
<legacy -patch-cli-script> 中的前缀 legacy- * 与将归档补丁应用到可引导 JAR 相关。此方法类似于将补丁应用到常规 JBoss EAP 分发。

您可以通过移除未使用的补丁内容，使用 JBoss EAP JAR Maven 插件配置中的 **legacy-patch-cleanup** 选项来减少可引导 JAR 的内存占用。选项可删除未使用的模块依赖项。在补丁配置文件中，此选项默认设置为 **false**。

legacy-patch-cleanup 选项会删除以下补丁内容：

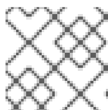
- **<JBOSS_HOME>/.installation/patches** 目录。

- 基础层中补丁模块的原始位置.
- 未使用的模块，这些模块由补丁添加且没有在现有模块图形或补丁的模块图中引用。
- 覆盖未在 **.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还与可引导的 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 7.4.4 GA 补丁
  - JBoss EAP XP 3.0.0 补丁

流程

1. 创建一个 CLI 脚本，以定义您要应用到可引导 JAR 的旧修补程序。脚本必须包含一个或多个 patch apply 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 **<legacy-patch-cli-script>** 元素中引用您的 CLI 脚本。
3. 重新构建可引导 JAR.

其他资源

- 有关下载 JBoss EAP MicroProfile Maven 存储库的信息，请参阅下载 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，请查看 *红帽客户门户网站* 中的技术预览功能支持范围。

第 9 章 REFERENCE

9.1. MICROPROFILE 配置参考

9.1.1. 默认 MicroProfile 配置属性

MicroProfile 配置规范 **默认定义三种 ConfigSource**。

ConfigSources 根据序号进行排序。如果以后的部署必须覆盖配置，则在较高序 **ConfigSource** 前覆盖了低序 **ConfigSource**。

表 9.1. 默认 MicroProfile 配置属性

ConfigSource	ordinal
系统属性	**400**
环境变量	**300**
类路径上找到的属性文件 **META-INF/microprofile-config.properties**	**100**

9.1.2. MicroProfile Config SmallRye ConfigSources

The **microprofile-config-smallrye** 项目定义了除默认的 MicroProfile **Config ConfigSources** 之外还可使用的更多 **ConfigSource**。

表 9.2. 其他 MicroProfile 配置属性

ConfigSource	ordinal
子系统 **中的 config-source**	**100**
来自 **目录的 ConfigSource**	**100**
来自类的 **ConfigSource**	**100**

没有为这些 **ConfigSources 指定显式的序数**。它们继承 MicroProfile 配置规范中找到的默认 ordinal 值。

9.2. MICROPROFILE 容错参考

9.2.1. MicroProfile 容错配置属性

SmallRye Fault Tolerance 规范除 MicroProfile 容错规范中定义的属性外，还定义了下列属性：

表 9.3. MicroProfile 容错配置属性

属性	默认值	Description
io.smallrye.faulttolerance.globalThreadPoolSize	**100**	容错机制使用的线程数。这不包括舱壁线程池。
io.smallrye.faulttolerance.timeoutExecutorThreads	**5**	用于调度超时的线程池的大小。

9.3. MICROPROFILE JWT 参考

9.3.1. MicroProfile 配置 JWT 标准属性

The **microprofile-jwt-smallrye** 子系统支持下列 MicroProfile 配置标准属性：

表 9.4. MicroProfile 配置 JWT 标准属性

属性	默认	Description
mp.jwt.verify.publickey	NONE	使用其中一个支持的格式编码公钥的字符串表示。如果您有 set **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。如果您有 set **mp.jwt.verify.publickey**，则不要设置。
mp.jwt.verify.issuer	NONE	被验证的任何 JWT 令牌的预期值 **都是声明**。

示例 **microprofile-config.properties** 配置：

```
mp.jwt.verify.publickey.location=META-INF/public.pem
mp.jwt.verify.issuer=jwt-issuer
```

9.4. MICROPROFILE OPENAPI 参考

9.4.1. MicroProfile OpenAPI 配置属性

除了标准的 MicroProfile OpenAPI 配置属性外，JBoss EAP 还支持以下额外的 MicroProfile OpenAPI 属性：这些属性可同时应用到全局范围和应用范围。

表 9.5. JBoss EAP 中的 MicroProfile OpenAPI 属性

属性	默认值	Description

属性	默认值	Description
mp.openapi.extensions.enabled	**true**	启用或禁用注册 OpenAPI 端点。 当设置为 **false** 时，将禁用 OpenAPI 文档的生成。您可以使用 config 子系统或在配置文件中为各个应用设置全局值，如 **/META-INF/microprofile-config.properties**。 您可以将此属性参数化为在不同环境中有选择地启用或禁用 **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一个 distinct **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还是相对于 OpenAPI 端点的位置。 需要服务器记录才能确保在存在非根上下文路径的情况下，OpenAPI 文档的用户可以相对于 OpenAPI 端点的主机构建到 REST 服务的有效 URL。 值 **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。 如果设置为 **false**，JBoss EAP XP 将生成服务器记录，包括可以访问部署的所有协议、主机和端口。