Red Hat JBoss Enterprise Application Platform 7.4

JBoss EAP XP 4.0.0 の使用

JBoss EAP XP 4.0.0 向け

Last Updated: 2024-03-04

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

JBoss EAP XP 4.0.0 向け

法律上の通知

Copyright © 2024 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

概要

本書では、JBoss EAP XP 4.0.0 で MicroProfile を使用する一般的な情報を提供します。

目次

多様性を受け入れるオープンソースの強化 4

JBOSS EAP ドキュメントへのフィードバック (英語のみ) 5

第1章 最新の MICROPROFILE 機能向けの JBOSS EAP XP 6
1.1. JBOSS EAP XP について 6
1.2. JBOSS EAP XP のインストール 6
1.3. JBOSS EAP XP マネージャー 7
1.4. JBOSS EAP XP MANAGER 4.0 COMMANDS 7
1.5. JBOSS EAP XP 4.0.0 ON JBOSS EAP 7.4.X のインストール 10
1.6. JBOSS EAP のアンインストール 11
1.7. JBOSS EAP XP の状態の表示 11
1.8. JBOSS EAP XP および JBOSS EAP 7.4.X ベースパッチのロールバック 12

第2章 MICROPROFILE について 13
2.1. MICROPROFILE CONFIG 13
2.2. MICROPROFILE FAULT TOLERANCE 14
2.3. MICROPROFILE HEALTH 15
2.4. MICROPROFILE JWT 16
2.5. MICROPROFILE METRICS 17
2.6. MICROPROFILE OPENAPI 18
2.7. MICROPROFILE OPENTRACING 18
2.8. MICROPROFILE REST クライアント 20
2.9. MICROPROFILE REACTIVE MESSAGING 21

第3章 JBOSS EAP での MICROPROFILE の管理 24
3.1. MICROPROFILE OPENTRACING の管理 24
3.2. MICROPROFILE CONFIG の設定 25
3.3. MICROPROFILE FAULT TOLERANCE 設定 27
3.4. MICROPROFILE HEALTH 設定 28
3.5. MICROPROFILE JWT 設定 31
3.6. MICROPROFILE METRICS 管理 31
3.7. MICROPROFILE OPENAPI の管理 33
3.8. MICROPROFILE REACTIVE MESSAGING の管理 36
3.9. スタンドアロンサーバー設定 37

第4章 JBOSS EAP の MICROPROFILE アプリケーションの開発 41
4.1. MAVEN および JBOSS EAP MICROPROFILE MAVEN リポジトリー 41
4.2. MICROPROFILE CONFIG の開発 44
4.3. MICROPROFILE FAULT TOLERANCE アプリケーションの開発 47
4.4. MICROPROFILE HEALTH の開発 52
4.5. MICROPROFILE JWT アプリケーションの開発 55
4.6. MICROPROFILE METRICS の開発 61
4.7. MICROPROFILE OPENAPI アプリケーションの開発 63
4.8. MICROPROFILE REST クライアントの開発 70

第5章 JBOSS EAP XP の OPENSHIFT イメージでマイクロサービスアプリケーションをビルドおよび実行 .. 74
5.1. アプリケーションのデプロイメントに向けた OPENSHIFT の準備 74
5.2. RED HAT コンテナーレジストリーへの認証の設定 75
5.3. JBOSS EAP XP の最新の OPENSHIFT イメージストリームおよびテンプレートのインポート 75
5.4. OPENSHIFT への JBOSS EAP XP SOURCE-TO-IMAGE (S2I) アプリケーションのデプロイ 76
5.5. JBOSS EAP XP SOURCE-TO-IMAGE (S2I) アプリケーションのデプロイメント後タスクの完了 78

第6章 機能のトリム **80**
6.1. 利用可能な JBOSS EAP レイヤー 80

第7章 RED HAT CODEREADY STUDIO での JBOSS EAP MICROPROFILE アプリケーション開発の有効化 . **86**
7.1. MICROPROFILE 機能を使用するための CODEREADY STUDIO の設定 86
7.2. CODEREADY STUDIO での MICROPROFILE クイックスタートの使用 87

第8章 起動可能な JAR **89**
8.1. 起動可能な JAR について 89
8.2. JBOSS EAP MAVEN プラグイン 89
8.3. 起動可能な JAR 引数 90
8.4. 起動可能な JAR サーバーの GALLEON レイヤーの指定 92
8.5. JBOSS EAP ベアメタルプラットフォームでの起動可能な JAR の使用 94
8.6. JBOSS EAP ベアメタルプラットフォームでの起動可能な JAR の作成 97
8.7. ビルド時に実行される CLI スクリプト 99
8.8. 実行時に CLI スクリプトを実行する 100
8.9. JBOSS EAP OPENSHIFT プラットフォームでの起動可能な JAR の使用 101
8.10. OPENSHIFT の起動可能な JAR の設定 104
8.11. OPENSHIFT でのアプリケーションでの CONFIGMAP の使用 105
8.12. 起動可能な JAR MAVEN プロジェクトの作成 107
8.13. 起動可能な JAR の JSON ロギングの有効化 109
8.14. 複数の起動可能な JAR インスタンスの WEB セッションデータストレージの有効化 114
8.15. CLI スクリプトを使用した起動可能な JAR の HTTP 認証の有効化 120
8.16. RED HAT SINGLE SIGN-ON での JBOSS EAP の起動可能な JAR アプリケーションのセキュア化 125
8.17. DEV モードでの起動可能な JAR のパッケージ化 130
8.18. サーバーアーティファクトのアップグレード 131
8.19. EAP7.4.GA 依存関係の更新 132
8.20. 起動可能な JAR への JBOSS EAP パッチの適用 133

第9章 JBOSS EAP での OPENID CONNECT **136**
9.1. JBOSS EAP での OPENID CONNECT 設定 136
9.2. ELYTRON-OIDC-CLIENT サブシステムの有効化 138
9.3. SECURING APPLICATIONS USING OPENID CONNECT WITH RED HAT SINGLE SIGN-ON 139
9.4. OPENID CONNECT を使用した JBOSS EAP 起動可能な JAR アプリケーションの開発 148

第10章 JBOSS EAP での可観測性 **161**
10.1. JBOSS EAP の OPENTELEMETRY 161
10.2. JBOSS EAP での OPENTELEMETRY 設定 161
10.3. JBOSS EAP での OPENTELEMETRY トレース 162
10.4. JBOSS EAP で OPENTELEMETRY トレースを有効にする 163
10.5. OPENTELEMETRY サブシステムの設定 163
10.6. JAEGER を使用してアプリケーションの OPENTELEMETRY トレースを観察する 164
10.7. OPENTELEMETRY トレースアプリケーションの開発 165

第11章 参照資料 **172**
11.1. MICROPROFILE CONFIG リファレンス 172
11.2. MICROPROFILE FAULT TOLERANCE リファレンス 172
11.3. MICROPROFILE JWT リファレンス 173
11.4. MICROPROFILE OPENAPI リファレンス 173
11.5. MICROPROFILE REACTIVE MESSAGING リファレンス 175
11.6. OPENID CONNECT リファレンス 181
11.7. OPENTELEMETRY リファレンス 195

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、Red Hat CTO である Chris Wright のメッセージ をご覧ください。

JBOSS EAP ドキュメントへのフィードバック (英語のみ)

エラーを報告したり、ドキュメントを改善したりするには、Red Hat Jira アカウントにログインし、課題を送信してください。Red Hat Jira アカウントをお持ちでない場合は、アカウントを作成するように求められます。

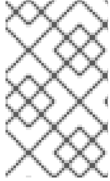
手順

1. このリンクをクリック してチケットを作成します。
2. **ドキュメント URL、セクション番号、課題の説明** を記入してください。
3. **Summary** に課題の簡単な説明を入力します。
4. **Description** に課題や機能拡張の詳細な説明を入力します。問題があるドキュメントのセクションへの URL を含めてください。
5. **Submit** をクリックすると、課題が作成され、適切なドキュメントチームに転送されます。

第1章 最新の MICROPROFILE 機能向けの JBOSS EAP XP

1.1. JBOSS EAP XP について

JBoss EAP XP (MicroProfile Expansion Pack) は、JBoss EAP XP マネージャーを使用して提供されるパッチストリームとして利用できます。

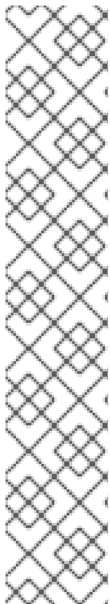


注記

JBoss EAP XP は、個別のサポートおよびライフサイクルポリシーに依存します。詳細は、JBoss Enterprise Application Platform expansion pack Support and Life Cycle Policies ページを参照してください。

JBoss EAP XP パッチは、以下の Eclipse MicroProfile 4.1 コンポーネントを提供します。

- MicroProfile Config
- MicroProfile Fault Tolerance
- MicroProfile Health
- MicroProfile JWT
- MicroProfile Metrics
- MicroProfile OpenAPI
- MicroProfile OpenTracing
- MicroProfile REST クライアント
- MicroProfile Reactive Messaging



注記

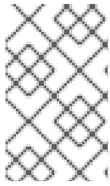
MicroProfile Reactive Messaging サブシステムは Red Hat AMQ Streams をサポートします。この機能は MicroProfile Reactive Messaging 2.0.1 API を実装し、Red Hat は JBoss EAP XP 4.0.0 のテクノロジープレビューとして機能を提供します。

Red Hat は、JBosscEAP で Red Hat AMQ Streams 2021.Q4 をテストしました。ただし、JBoss EAP XP 4.0.0 でテストされた最新の Red Hat AMQ Streams バージョンについては、Red Hat JBoss Enterprise Application Platform supported configurations ページを参照してください。

テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。Red Hat は、実稼働環境でこれらを使用することを推奨していません。テクノロジープレビュー機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

1.2. JBOSS EAP XP のインストール

JBoss EAP XP をインストールする場合は、JBoss EAP XP パッチが JBoss EAP のバージョンと互換性があることを確認してください。JBoss EAP XP 4.0.x パッチは JBoss EAP 7.4 リリースと互換性があります。



注記

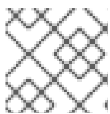
JBoss EAP XP は、XP マネージャーおよび EAP アーカイブを使用するか、JBoss EAP XP OpenShift Container イメージを使用してインストールできます。EAP RPM に JBoss EAP XP をインストールすることはできません。

関連情報

- 最新の JBoss EAP リリースに最新の JBoss EAP XP パッチをインストールする方法は、Installing JBoss EAP XP 4.0.0 on JBoss EAP 7.4.x を参照してください。

1.3. JBOSS EAP XP マネージャー

JBoss EAP XP マネージャーは、**製品のダウンロードページ**からダウンロードできる実行可能な **jar** ファイルです。JBoss EAP XP マネージャーを使用して、JBoss EAP XP パッチストリームから JBoss EAP XP パッチを適用します。このパッチには MicroProfile 4.1 実装と、これらの MicroProfile 4.1 実装のバグ修正が含まれます。

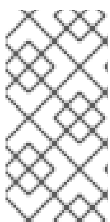


注記

管理コンソールを使用して JBoss EAP XP パッチを管理できません。

引数を指定せずに JBoss EAP XP マネージャーを実行したり、**help** コマンドを実行すると、使用できるコマンドのリストと、そのコマンドの実行内容が表示されます。

help コマンドでマネージャーを実行して、利用可能な引数の詳細情報を取得します。



注記

JBoss EAP XP マネージャーのコマンドのほとんどは、**--jboss-home** 引数を取り、JBoss EAP XP サーバーを参照して JBoss EAP XP パッチストリームを管理します。これを省略する場合は、**JBOSS_HOME** 環境変数でサーバーへのパスを指定します。**--jboss-home** は環境変数よりも優先されます。

1.4. JBOSS EAP XP MANAGER 4.0 COMMANDS

JBoss EAP XP マネージャー 4.0 は、JBoss EAP XP パッチストリームを管理するためのさまざまなコマンドを提供します。

以下のコマンドが提供されます。

patch-apply

このコマンドを使用して JBoss EAP インストールにパッチを適用します。
patch-apply コマンドは、**patch apply** 管理 CLI コマンドに似ています。**patch-apply** コマンドは、ツールを使用してパッチを適用するために必要な引数のみを受け入れます。他の **patch apply** 管理 CLI コマンド引数にデフォルト値を使用します。

patch-apply コマンドを使用して、サーバーで有効なパッチストリームにパッチを適用できます。コマンドを使用して、ベースサーバーのパッチと XP パッチの両方を適用することもできます。

patch-apply コマンドの使用例

```
$ 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
```

XP パッチを適用するとき、JBoss EAP XP マネージャー 40 は検証を実行し、パッチとパッチストリームの不一致を防ぎます。以下の例は、誤った組み合わせを示しています。

- XP 4.0 パッチストリームが設定されたサーバーに JBoss EAP XP 3.0 パッチをインストールすると、以下のエラーが発生します。

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

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

- JBoss EAP XP 4.0.0 パッチストリーム用に設定されていないサーバーに JBoss EAP XP 4.0.0 パッチをインストールすると、以下のエラーが発生します。

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

いずれの場合も、サーバーに変更が加えられません。

remove

このコマンドを使用して、JBoss EAP サーバーから JBoss EAP XP パッチストリーム設定を削除します。

remove コマンドの使用例

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

setup

このコマンドを使用して、JBoss EAP XP パッチストリームにクリーンな JBoss EAP サーバーを設定します。
setup コマンドを使用すると、JBoss EAP XP マネージャーは以下の操作を実行します。

- JBoss EAP XP 4.0.0 パッチストリームを有効にします。

- **--base-patch** および **--xp-patch** 属性を使用して指定されたパッチを適用します。
- **standalone-microprofile.xml** および **standalone-microprofile-ha.xml** 設定ファイルをサーバー設定ディレクトリーにコピーします。
  古い設定ファイルがすでにインストールされている場合、新しいファイルは **standalone-microprofile-yyyyMMdd-HHmmss.xml** などのターゲット設定ディレクトリーにタイムスタンプ付きコピーとして保存されます。

  --jboss-config-directory 引数を使用してターゲットディレクトリーを設定できます。

setup コマンドの使用例

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

status

このコマンドを使用して、JBoss EAP XP サーバーの現在の状態を見つけます。status コマンドは、以下の情報を返します。

- JBoss EAP XP ストリームの状態
- 現在の状態によるサポートポリシーの変更
- JBoss EAP XP のメジャーバージョン。
- パッチストリームと累積パッチ ID を有効にしました。
- 状態を変更するのに利用できる JBoss EAP XP マネージャーコマンド

status コマンドの使用例

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

upgrade

このコマンドを使用して、JBoss EAP サーバーで古い JBoss EAP XP パッチストリームを JBoss EAP サーバーの最新のパッチストリームにアップグレードします。
upgrade コマンドを使用すると、JBoss EAP XP マネージャーは以下の操作を実行します。

- サーバーで古いパッチストリームを有効にするファイルのバックアップを作成します。
- JBoss EAP XP 4.0 パッチストリームを有効にします。
- **--base-patch** および **--xp-patch** 属性を使用して指定されたパッチを適用します。
- **standalone-microprofile.xml** および **standalone-microprofile-ha.xml** 設定ファイルをサーバー設定ディレクトリーにコピーします。古い設定ファイルがすでにインストールされている場合、新しいファイルは **standalone-microprofile-yyyyMMdd-HHmmss.xml などのターゲット設定ディレクトリーにタイムスタンプ付きコピーとして保存** されます。
- 問題が発生した場合、JBoss EAP XP マネージャーは作成したバックアップから以前のパッチストリームを復元しようとします。
  --jboss-config-directory 引数を使用してターゲットディレクトリーを設定できます。

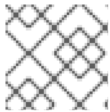
upgrade コマンドの使用例:

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

1.5. JBOSS EAP XP 4.0.0 ON JBOSS EAP 7.4.X のインストール

JBoss EAP XP 4.0.0 を JBoss EAP 7.4 ベースサーバーにインストールします。

JBoss EAP XP マネージャー 4.0.0 を使用して JBoss EAP XP 4.0.0 パッチストリームを管理します。



注記

JBoss EAP XP 4.0.0 は JBoss EAP 7.4.x で認定されています。

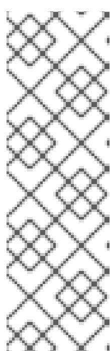
前提条件

- **製品のダウンロード**ページから以下のファイルをダウンロードしている。
  - **jboss-eap-xp-4.0.0-manager.jar** ファイル (JBoss EAP XP マネージャー 4.0)
  - JBoss EAP 7.4 サーバーアーカイブファイル
  - JBoss EAP XP 4.0.0 パッチ

手順

1. ダウンロードした JBoss EAP 7.4 サーバーアーカイブファイルを JBoss EAP インストールのパスに展開します。
2. 以下のコマンドを使用して JBoss EAP XP 4.0 パッチストリームを管理するために、JBoss EAP XP マネージャー 4.0.0 を設定します。

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



注記

JBoss EAP XP 4.0.0 パッチを同時に適用することができます。**--xp-patch** 引数を使用して JBoss EAP XP 4.0.0 パッチへのパスを含めます。

例:

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

これで、JBoss EAP XP 4.0.0 パッチストリームを管理できるようになりました。

3. オプション: **--xp-patch** 引数を使用して JBoss EAP XP 4.0.0 パッチを JBoss EAP サーバーに適用していない場合は、JBoss EAP XP マネージャー 4.0.0 **patch-apply** コマンドを使用して JBoss EAP XP 4.0.0 パッチを適用 します。

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

patch-apply コマンドは、**patch apply** 管理 CLI コマンドに似ています。**patch apply** 管理 CLI コマンドを使用して、パッチを適用することもできます。

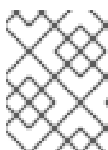
JBoss EAP サーバーが JBoss EAP XP 4.0.0 パッチが適用されたため、JBoss EAP XP 4.0.0 パッチストリームを管理できるようになりました。

関連情報

- JBoss EAP XP manager 4.0 commands

1.6. JBOSS EAP のアンインストール

JBoss EAP XP をアンインストールすると、JBoss EAP XP 4.0.0 パッチストリームと MicroProfile 4.1 機能の有効化に関連するすべてのファイルが削除されます。アンインストールプロセスは、ベースサーバーのパッチストリームまたは機能には影響しません。



注記

アンインストールプロセスでは、JBoss EAP XP パッチストリームを有効にしたときに JBoss EAP XP パッチに追加した設定ファイルなどは削除されません。

手順

- 以下のコマンドを実行して JBoss EAP XP 4.0.0 をアンインストールします。

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

MicroProfile 4.1 機能を再度インストールするには、再度 **setup** コマンドを実行してパッチストリームを有効にし、JBoss EAP XP パッチを適用して MicroProfile 4.1 モジュールを追加します。

1.7. JBOSS EAP XP の状態の表示

status コマンドを使用して、以下の情報を表示できます。

- JBoss EAP XP ストリームの状態
- 現在の状態によるサポートポリシーの変更
- JBoss EAP XP のメジャーバージョン。
- パッチストリームと、その累積パッチ ID を有効にしている。
- 状態を変更するのに利用できる JBoss EAP XP マネージャーコマンド

JBoss EAP XP は、以下のいずれかの状態になります。

Not set up

JBoss EAP はクリーンな状態で、JBoss EAP XP は設定されていません。

Set up

JBoss EAP で JBoss XP がセットアップされています。XP パッチストリームのバージョンは、ユーザーが CLI を使用して判断できるので表示されません。

Inconsistent

JBoss EAP XP に関連するファイルは、一貫性のない状態です。これはエラー状態であるため、通常

は発生しません。このエラーが発生する場合は、JBoss EAP XP のアンインストールのトピックで説明されているように JBoss EAP XP マネージャーを削除し、**setup** コマンドを使用して JBoss EAP XP を再度インストールします。

手順

- 以下のコマンドを実行して、JBoss EAP XP の状態を表示します。

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

関連情報

- JBoss EAP のアンインストール
- Installing JBoss EAP XP 4.0.0 on JBoss EAP 7.4.x

1.8. JBOSS EAP XP および JBOSS EAP 7.4.X ベースパッチのロールバック

管理 CLI を使用して、以前に適用された JBoss EAP XP パッチまたは JBoss EAP 7.4.x ベースパッチにロールバックできます。

関連情報

- JBoss EAP XP パッチまたは JBoss EAP 7.4.x ベースパッチのロールバックに関する詳細は、Rolling back a patch using the management CLI を参照してください。

第2章 MICROPROFILE について

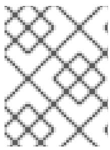
2.1. MICROPROFILE CONFIG

2.1.1. JBoss EAP での MicroProfile Config

設定データは動的に変更でき、アプリケーションはサーバーを再起動せずに最新の設定情報にアクセスできる必要があります。

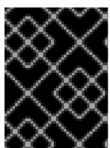
MicroProfile Config は設定データのポータブルな外部化を実現します。つまり、アプリケーションとマイクロサービスを、変更または再パッケージ化せずに複数の環境で実行するように設定できます。

MicroProfile Config 機能は、SmallRye Config を使用して JBoss EAP に実装され、**microprofile-config-smallrye** サブシステムによって提供されます。



注記

MicroProfile Config は JBoss EAP XP でのみサポートされます。これは JBoss EAP ではサポートされません。



重要

独自の Config 実装を追加する場合は、Config インターフェイスの最新バージョンのメソッドを使用する必要があります。

関連情報

- MicroProfile Config
- SmallRye Config
- Config implementations

2.1.2. MicroProfile Config でサポートされる MicroProfile Config ソース

MicroProfile Config 設定プロパティーは、さまざまな場所から取得でき、形式が異なる場合があります。これらのプロパティーは ConfigSources によって提供されます。ConfigSources は **org.eclipse.microprofile.config.spi.ConfigSource** インターフェイスの実装です。

MicroProfile Config 仕様は、設定値を取得するために、以下のデフォルト **ConfigSource** 実装を提供します。

- **System.getProperties()**
- **System.getenv()**
- クラスパス上のすべての **META-INF/microprofile-config.properties**。

microprofile-config-smallrye サブシステムは、設定値を取得するために **ConfigSource** リソースの追加タイプをサポートします。以下のリソースから設定値を取得することもできます。

- **microprofile-config-smallrye/config-source** 管理リソースでのプロパティー
- ディレクトリー内のファイル

- **ConfigSource** クラス
- **ConfigSourceProvider** クラス

関連情報

- [org.jboss.resteasy.microprofile.config.BaseServletConfigSource]

2.2. MICROPROFILE FAULT TOLERANCE

2.2.1. MicroProfile Fault Tolerance 仕様について

MicroProfile Fault Tolerance 仕様は、分散したマイクロサービスに特有のエラーに対応するストラテジーを定義します。

MicroProfile Fault Tolerance 仕様は、エラーを処理する以下のストラテジーを定義します。

Timeout
: 実行が終了べき時間を定義します。タイムアウトを定義すると、実行を永久に待機できなくなります。

Retry
: 失敗した実行を再試行する基準を定義します。

Fallback
: 実行に失敗した場合の代替を指定します。

CircuitBreaker
: 一時的に停止するまでの実行試行回数を定義します。遅延の長さを定義すると、実行を再開することができます。

Bulkhead
: システムの一部で障害を分離して、残りのシステムを機能させせます。

Asynchronous
: 別のスレッドでクライアント要求を実行します。

関連情報

- MicroProfile Fault Tolerance Specification

2.2.2. JBoss EAP での MicroProfile Fault Tolerance

microprofile-fault-tolerance-smallrye サブシステムは、JBoss EAP での MicroProfile Fault Tolerance のサポートを提供します。このサブシステムは、JBoss EAP XP ストリームでのみ利用できます。

microprofile-fault-tolerance-smallrye サブシステムはインターセプターバインディングに以下のアノテーションを提供します。

- **@Timeout**
- **@Retry**
- **@Fallback**
- **@CircuitBreaker**

- **@Bulkhead**
- **@Asynchronous**

これらのアノテーションはクラスレベルまたはメソッドレベルでバインドできます。クラスにバインドされたアノテーションは、そのクラスのすべてのビジネスメソッドに適用されます。

以下のルールはバインディングインターセプターに適用されます。

- コンポーネントクラスがクラスレベルのインターセプターバインディングを宣言または継承する場合、以下の制限が適用されます。
  - クラスは final を宣言することはできません。
  - クラスには static、private、または final メソッドを含めることはできません。
- コンポーネントクラスの静的ではない非プライベートメソッドがメソッドレベルのインターセプターバインディングを宣言する場合、メソッドやコンポーネントクラスも final 宣言されません。

フォールトトレランス操作には以下の制限があります。

- フォールトトレランスインターセプターバインディングは bean クラスまたは bean クラスメソッドに適用する必要があります。
- 呼び出し時、呼び出しは Jakarta Contexts and Dependency Injection 仕様で定義されているビジネスメソッド呼び出しである必要があります。
- 以下の条件が両方とも true の場合、操作はフォールトトレランスと見なされません。
  - メソッド自体は、フォールトトレランスインターセプターにバインドされません。
  - メソッドが含まれるクラスは、フォールトトレランスインターセプターにバインドされません。

microprofile-fault-tolerance-smallrye サブシステムは、MicroProfile Fault Tolerance が提供する設定オプションに加え、以下の設定オプションを提供します。

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

関連情報

- MicroProfile Fault Tolerance 仕様
- SmallRye Fault Tolerance プロジェクト

2.3. MICROPROFILE HEALTH

2.3.1. JBoss EAP での MicroProfile Health

JBoss EAP には SmallRye Health コンポーネントが含まれており、これを使用して JBoss EAP インスタンスが想定どおりに応答しているかどうかを判断できます。この機能はデフォルトで有効になります。

MicroProfile Health は、JBoss EAP をスタンドアロンサーバーとして実行している場合のみ利用できます。

MicroProfile Health 仕様は、以下のヘルスチェックを定義します。

Readiness
: アプリケーションがリクエストを処理する準備ができているかどうかを決定します。**@Readiness** アノテーションは、このヘルスチェックを提供します。

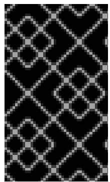
Liveness
: アプリケーションが実行されているかどうかを決定します。**@Liveness** アノテーションは、このヘルスチェックを提供します。

Startup
: アプリケーションがすでに開始されているかどうかを判別します。アノテーション **@Startup** は、このヘルスチェックを提供します。

@Health アノテーション は MicroProfile Health 3.0 で削除されました。

MicroProfile Health 3.1 には、新しい **Startup** ヘルスチェックプローブが含まれています。

MicroProfile Health 3.1 における互換性の重大な変更に関する詳細は、Release Notes for MicroProfile Health 3.1 を参照してください。



重要

:empty-readiness-checks-status、**:empty-liveness-checks-status**、および **:empty-startup-checks-status** 管理属性は、**readiness**、**liveness**、または **startup** プローブが定義されていない場合のグローバルステータスを指定します。

関連情報

- プローブが定義されていない場合のグローバルステータス
- smallrye Health
- MicroProfile Health
- カスタムヘルスチェックの例

2.4. MICROPROFILE JWT

2.4.1. JBoss EAP での MicroProfile JWT 統合

サブシステム **microprofile-jwt-smallrye** は、JBoss EAP で MicroProfile JWT 統合を提供します。

以下の機能は **microprofile-jwt-smallrye** サブシステムによって提供されます。

- MicroProfile JWT セキュリティーを使用するデプロイメントの検出。
- MicroProfile JWT のサポートの有効化。

サブシステムには設定可能な属性やリソースが含まれません。

org.eclipse.microprofile.jwt.auth.api モジュールは、**microprofile-jwt-smallrye** サブシステムの他に、JBoss EAP で MicroProfile JWT 統合を提供します。

関連情報

- SmallRye JWT

2.4.2. 従来のデプロイメントと MicroProfile JWT デプロイメントの相違点

MicroProfile JWT デプロイメントは、従来の JBoss EAP デプロイメントなどの管理された SecurityDomain リソースに依存しません。代わりに、仮想 SecurityDomain が作成され、MicroProfile JWT デプロイメント全体で使用されます。

MicroProfile JWT デプロイメントは MicroProfile Config プロパティーと **microprofile-jwt-smallrye** サブシステム内で完全に設定されるため、仮想 SecurityDomain はデプロイメントの他の管理設定を必要としません。

2.4.3. JBoss EAP での MicroProfile JWT アクティベーション

MicroProfile JWT は、アプリケーションに **auth-method** の有無に基づいてアプリケーションに対してアクティベートされます。

MicroProfile JWT 統合は、以下のようにアプリケーションに対してアクティベートされます。

- デプロイメントプロセスの一環として、JBoss EAP はアプリケーションアーカイブで **auth-method** の存在をスキャンします。
- **auth-method** 存在し、**MP-WT** として定義されている場合は、MicroProfile JWT 統合がアクティベートされます。

auth-method は、以下のファイルのいずれかまたは両方で指定できます。

- **javax.ws.rs.core.Application** を拡張するクラスを含むファイル。**@LoginConfig** アノテーション付き。
- **web.xml** 設定ファイル

auth-method がアノテーションを使用して、および web.xml 設定ファイルの両方に定義されている場合は、**web.xml** 設定ファイルの定義が使用されます。

2.4.4. JBoss EAP での MicroProfile JWT の制限

JBoss EAP の MicroProfile JWT 実装にはいくつかの制限があります。

JBoss EAP には、MicroProfile JWT 実装の制限があります。

- MicroProfile JWT 実装は、**mp.jwt.verify.publickey** プロパティーで提供された JSON Web Key Set(JWKS) からの最初の鍵のみを解析します。したがって、トークンが 2 つ目の鍵または 2 つ目の鍵の後に署名されるように要求すると、トークンの検証に失敗し、トークンを含むリクエストは承認されません。
- JWKS の base64 エンコードはサポートされていません。

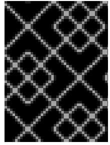
いずれの場合も、**mp.jwt.verify.publickey.location** 設定プロパティーを使用する代わりに、クリアーテキスト JWKS を参照できます。

2.5. MICROPROFILE METRICS

2.5.1. JBoss EAP での MicroProfile Metrics

JBoss EAP には SmallRye Metrics コンポーネントが含まれています。JBoss EAP では、**microprofile-metrics-smallrye** サブシステムを使用して MicroProfile Metrics 機能を提供する SmallRye Metrics コンポーネントを利用できます。

microprofile-metrics-smallrye サブシステムは JBoss EAP インスタンスのモニタリングデータを提供します。サブシステムはデフォルトで有効になっています。



重要

microprofile-metrics-smallrye サブシステムは、スタンドアロン設定でのみ有効になります。

関連情報

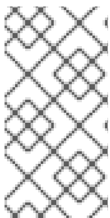
- SmallRye Metrics
- MicroProfile Metrics

2.6. MICROPROFILE OPENAPI

2.6.1. JBoss EAP での MicroProfile OpenAPI

MicroProfile OpenAPI は、**microprofile-openapi-smallrye** サブシステムを使用して JBoss EAP に統合されます。

MicroProfile OpenAPI 仕様は、OpenAPI 3.0 ドキュメントを提供する HTTP エンドポイントを定義します。OpenAPI 3.0 ドキュメントでは、ホストの REST サービスについて説明します。OpenAPI エンドポイントは、設定されたパス (例: http://localhost:8080/openapi) を使用してデプロイメントに関連付けられたホストのルートに対して登録されます。



注記

現在、仮想ホストの OpenAPI エンドポイントは単一デプロイメントのみを文書化できます。同じ仮想ホストの異なるコンテキストパスで登録された複数のデプロイメントで OpenAPI を使用するには、各デプロイメントは個別のエンドポイントパスを使用する必要があります。

OpenAPI エンドポイントはデフォルトで YAML ドキュメントを返します。Accept HTTP ヘッダーまたは format クエリーパラメーターを使用して JSON ドキュメントをリクエストすることもできます。

指定のアプリケーションの Undertow サーバーまたはホストが HTTPS リスナーを定義する場合、OpenAPI ドキュメントも HTTPS を使用して利用できます。たとえば、HTTPS のエンドポイントは https://localhost:8443/openapi です。

2.7. MICROPROFILE OPENTRACING

2.7.1. MicroProfile OpenTracing

サービス境界全体でリクエストをトレースする機能は、ライフサイクル中にリクエストが複数のサービスを通過するマイクロサービス環境で特に重要となります。

MicroProfile OpenTracing 仕様は、CDI-bean アプリケーション内の OpenTacing 対応の **Tracer** オインターフェイスにアクセスするための、動作および API を定義します。**Tracer** インターフェイスは JAX-RS アプリケーションを自動的にトレースします。

動作は、送受信リクエストに対してどのように Open Tracing Spans が自動的に作成されるかを指定します。API は、指定のエンドポイントのトレースをどのように明示的に無効または有効にするかを定義します。

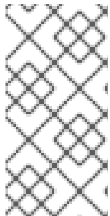
関連情報

- MicroProfile OpenTracing 仕様の詳細は、MicroProfile OpenTracing のドキュメント を参照してください。
- **Tracer** インターフェイスの詳細は、**Tracer** の java ドキュメント を参照してください。

2.7.2. JBoss EAP での MicroProfile OpenTracing

microprofile-opentracing-smallrye サブシステムを使用して、Jakarta EE アプリケーションの分散トレーシングを設定できます。このサブシステムは SmallRye OpenTracing コンポーネントを使用して JBoss EAP の MicroProfile OpenTracing 機能を提供します。

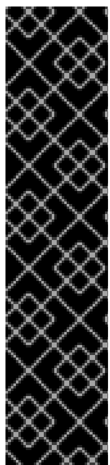
MicroProfile OpenTracing 2.0 は、アプリケーションのリクエストのトレースをサポートします。デフォルトの Jaeger Java Client トレーサーや、管理 CLI または管理コンソールで JBoss EAP 管理 API を使用して、Jakarta EE で一般的に使用されるコンポーネントのインスツルメンテーションライブラリーのセットを設定できます。



注記

JBoss EAP サーバーに自動的にデプロイされた各 WAR は、独自の **Tracer** インスタンスを持ちます。EAR 内の各 WAR は個別の WAR として扱われ、各 WAR には独自の **Tracer** インスタンスがあります。デフォルトでは、Jaeger Client と使用されるサービス名はデプロイメントの名前から派生し、通常これは WAR ファイル名になります。

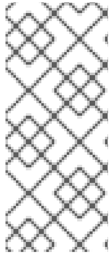
microprofile-opentracing-smallrye サブシステム内でシステムプロパティーまたは環境変数を設定して Jaeger Java Client を設定できます。



重要

システムプロパティーおよび環境変数を使用した Jaeger Client トレーサーの設定はテクノロジープレビューとして提供されます。Jaeger Client トレーサーに関連するシステムプロパティーおよび環境変数は、今後のリリースで変更されて、相互互換性がなくなる可能性があります。

テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。Red Hat は、実稼働環境でこれらを使用することを推奨していません。テクノロジープレビュー機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。



注記

デフォルトでは、Jaeger Client for Java のプローブ的なサンプリングストラテジーは **0.001** に設定されています。つまり、サンプルされるのは、約 1000 トレースつき 1 つとなります。すべてのリクエストのサンプルを取るには、システムプロパティー **JAEGER_SAMPLER_TYPE** を **const** に設定し、**JAEGER_SAMPLER_PARAM** を **1** に設定します。

関連情報

- SmallRye OpenTracing 機能の詳細は、SmallRye OpenTracing コンポーネント を参照してください。
- デフォルトのトレーサーの詳細は、Jaeger Java Client を参照してください。
- **Tracer** インターフェイスの詳細は、**Tracer** の java ドキュメント を参照してください。
- デフォルトのトレーサーの上書きおよび Jakarta Contexts and Dependency Injection bean のトレースに関する詳細は、**開発ガイド**の Eclipse MicroProfile OpenTracing を使用したリクエストのトレース を参照してください。
- Jaeger Client の設定に関する詳細は、Jaeger ドキュメント を参照してください。
- 有効なシステムプロパティーの詳細は、Jaeger ドキュメントの Configuration via Environment を参照してください。

2.8. MICROPROFILE REST クライアント

2.8.1. MicroProfile REST クライアント

JBoss EAP XP 4.0.0 は、HTTP 上で RESTful サービスを呼び出すために型安全なアプローチを提供するために Jakarta RESTful Web Services 2.1.6 クライアント API 上にビルドされる MicroProfile REST クライアント 2.0 をサポートします。MicroProfile Type Safe REST クライアントは、Java インターフェイスとして定義されます。MicroProfile REST クライアントでは、実行可能コードでクライアントアプリケーションを作成できます。

MicroProfile REST クライアントを使用して以下の機能を利用します。

- 直感的な構文
- プロバイダーのプログラムによる登録
- プロバイダーの宣言的登録
- ヘッダーの宣言的仕様
- **ResponseExceptionMapper**
- Jakarta Contexts and Dependency Injection の統合
- サーバー向けイベント (SSE) へのアクセス

関連情報

- MicroProfile REST クライアントと Jakarta RESTful Web Services 構文の比較

- MicroProfile REST クライアントでのプロバイダーのプログラムによる登録
- MicroProfile REST クライアントでのプロバイダーの宣言的登録
- MicroProfile REST クライアントでのヘッダーの宣言型仕様
- MicroProfile REST クライアントの ResponseExceptionMapper
- MicroProfile REST クライアントでのコンテキスト依存関係の挿入

2.8.2. **resteasy.original.webapplicationexception.behavior** MicroProfile Config プロパティー

MicroProfile Config は、開発者がアプリケーションとマイクロサービスを設定して、それらのアプリケーションを変更または再パッケージ化することなく、複数の環境で実行できるようにするために使用できる仕様の名前です。以前は、MicroProfile Config はテクノロジープレビューとして JBoss EAP で利用可能でしたが、その後削除されました。MicroProfile Config は、JBoss EAP XP でのみ使用できるようになりました。

resteasy.original.webapplicationexception.behavior MicroProfile Config プロパティーの定義
resteasy.original.webapplicationexception.behavior パラメーターは、**web.xml** サーブレットプロパティーまたはシステムプロパティーのいずれかとして設定できます。**web.xml** のそのようなサーブレットプロパティーの例を次に示します。

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

MicroProfile Config を使用して、他の RESTEasy プロパティーを設定することもできます。

関連情報

- JBoss EAP XP での MicroProfile Config の詳細は、**Understand MicroProfile** を参照してください。
- MicroProfile REST クライアントの詳細は、MicroProfile REST Client を参照してください。
- RESTEasy の詳細は、Jakarta RESTful Web Services Request Processing を参照してください。

2.9. MICROPROFILE REACTIVE MESSAGING

2.9.1. MicroProfile Reactive Messaging

JBoss EAP XP 4.0.0 にアップグレードする際に、リアクティブメッセージングエクステンションおよびサブシステムが含まれる最新バージョンの MicroProfile Reactive Messaging を有効化できます。

リアクティブストリームは、処理プロトコルと標準とともに一連のイベントデータであり、バッファリングなしで非同期境界 (スケジューラーなど) を超えてプッシュされます。イベントは、たとえば、天気予報アプリケーションでスケジュールされ、繰り返される温度チェックの場合があります。リアクティブストリームの主な利点は、さまざまなアプリケーションと実装のシームレスな相互運用性です。

リアクティブメッセージングは、イベント駆動型、データストリーミング、およびイベントソーシングアプリケーションをビルドするためのフレームワークを提供します。リアクティブメッセージングによ

り、あるアプリケーションから別のアプリケーションへのイベントデータ (リアクティブストリーム) の継続的かつスムーズな交換が実現します。MicroProfile Reactive Messaging を使用して、リアクティブストリームを介した非同期メッセージングを行うことができます。これにより、アプリケーションは、たとえば Apache Kafka などの他のアプリケーションと対話できます。

MicroProfile Reactive Messaging のインスタンスを最新バージョンにアップグレードした後、次の操作を実行できます。

- Apache Kafka データストリーミングプラットフォーム用の MicroProfileReactiveMessaging を使用してサーバーをプロビジョニングします。
- 最新のリアクティブメッセージング API を介して、メモリー内および Apache Kafka トピックでサポートされるリアクティブメッセージングと対話する。
- MicroProfile Metrics を使用して、指定のチャネルでストリーミングされるメッセージの数を把握する。

関連情報

- Apache Kafka の詳細は、What is Apache Kafka? を参照してください。

2.9.2. MicroProfile リアクティブメッセージングコネクター

コネクターを使用して、MicroProfile Reactive Messaging を多数の外部メッセージングシステムと統合できます。MicroProfile for JBoss EAP には、Apache Kafka コネクターが付属しています。Eclipse MicroProfile Config 仕様を使用して、コネクターを設定します。

Apache Kafka コネクターと組み込まれたレイヤー

MicroProfile Reactive Messaging には、MicroProfile Config で設定できる Kafka コネクターが含まれています。Kafka コネクターには、**microprofile-reactive-messaging-kafka** レイヤーと **microprofile-reactive-messaging** レイヤーが組み込まれています。**microprofile-reactive-messaging** レイヤーは、コアの MicroProfile Reactive Messaging 機能を提供します。

表2.1 Reactive Messaging と Apache Kafka コネクター Galleon レイヤー

<table>
<tr><th>layer</th><th>定義</th></tr>
<tr><td>microprofile-reactive-streams-operators</td><td>MicroProfile Reactive Streams Operators API を提供し、モジュールの実装をサポートします。
SmallRye エクステンションとサブシステムを備えた MicroProfile Reactive Streams Operators が含まれています。
cdi レイヤーに依存します。
cdi は、Jakarta Contexts and Dependency Injection の略です。@Inject 機能を追加するサブシステムを提供します。</td></tr>
</table>

layer	定義
microprofile-reactive-messaging	• MicroProfile Reactive Messaging API を提供し、モジュールの実装をサポートします。 • SmallRye エクステンションとサブシステムを備えた MicroProfile が含まれています。 • **microprofile-config** と **microprofile-reactive-streams-operators** レイヤーに依存します。
microprofile-reactive-messaging-kafka	• MicroProfile Reactive Messaging が Kafka と対話できるようにする Kafka コネクターモジュールを提供します。 • **microprofile-reactive-messaging** レイヤーに依存します。

2.9.3. Apache Kafka イベントストリーミングプラットフォーム

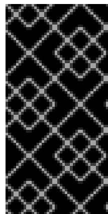
Apache Kafka は、レコードのストリームをリアルタイムでパブリッシュ、登録、保存、および処理できるオープンソースの分散イベント (データ) ストリーミングプラットフォームです。複数のソースからのイベントストリームを処理し、それらを複数のコンシューマーに配信して、大量のデータをポイント A から Z、およびその他の場所にすべて同時に移動します。MicroProfile Reactive Messaging は、Apache Kafka を使用して、これらのイベントレコードをわずか 2 マイクロ秒で配信し、分散したフォールトトレラントクラスターに安全に保存し、チーム定義のゾーンまたは地理的地域全体で利用できるようにします。

関連情報

- What is Apache Kafka?
- Red Hat OpenShift Streams for Apache Kafka
- Red Hat AMQ

第3章 JBOSS EAP での MICROPROFILE の管理

3.1. MICROPROFILE OPENTRACING の管理



重要

REST 呼び出し用にエクスポートされた重複トレースが表示される場合は、**microprofile-opentracing-smallrye** サブシステムを無効にします。**microprofile-opentracing-smallrye** を無効にする方法については Removing the microprofile-opentracing-smallrye subsystem を参照してください。

3.1.1. MicroProfile Open Tracing の有効化

以下の管理 CLI コマンドを使用してサーバー設定にサブシステムを追加し、サーバーインスタンスに対して MicroProfile Open Tracing 機能をグローバルに有効にします。

手順

1. 以下の管理コマンドを使用して **microprofile-opentracing-smallrye** サブシステムを有効にします。

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

2. 変更を反映するためにサーバーをリロードします。

```
reload
```

3.1.2. microprofile-opentracing-smallrye サブシステムの削除

microprofile-opentracing-smallrye サブシステムは、デフォルトの JBoss EAP 7.4 設定に含まれています。このサブシステムは、JBoss EAP 7.4 の MicroProfile OpenTracing 機能を提供します。MicroProfile OpenTracing を有効にしてシステムメモリーやパフォーマンスが低下した場合は、**microprofile-opentracing-smallrye** サブシステムを無効にすることができます。

管理 CLI で **remove** 操作を使用すると、指定のサーバーで MicroProfile OpenTracing 機能をグローバルに無効にできます。

手順

1. **microprofile-opentracing-smallrye** サブシステムを削除します。

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

2. 変更を反映するためにサーバーをリロードします。

```
reload
```

3.1.3. Jaeger のインストール

docker を使用して Jaeger をインストールします。

前提条件

- **docker** がインストールされている。

手順

1. CLI で以下のコマンドを実行して **docker** を使用して Jaeger をインストールします。

```
$ 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 CONFIG の設定

3.2.1. ConfigSource 管理リソースでのプロパティーの追加

プロパティーは管理リソースとして **config-source** サブシステムに直接保存できます。

手順

- ConfigSource を作成し、プロパティーを追加します。

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

3.2.2. ディレクトリーを ConfigSources として設定

プロパティーがファイルとしてディレクトリーに保存されている場合、file-name はプロパティーの名前で、ファイルの内容はプロパティーの値になります。

手順

1. ファイルを保存するディレクトリーを作成します。

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

2. ディレクトリーに移動します。

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

3. プロパティー **name** の値を保存するファイル **name** を作成します。

```
$ touch name
```

4. プロパティーの値をファイルに追加します。

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

5. ファイル名がプロパティーであり、プロパティーの値が含まれるファイルが含まれる ConfigSource を作成します。

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

これにより、以下の XML 設定が以下のようになります。

```
<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. ConfigSource クラスからの ConfigSource の取得

カスタムの **org.eclipse.microprofile.config.spi.ConfigSource** 実装クラスを作成および設定して、設定値のソースを提供することができます。

手順

- 以下の管理 CLI コマンドは、**org.example** という名前の JBoss モジュールによって提供される、**org.example.MyConfigSource** という名前の実装クラスの **ConfigSource** を作成します。**org.example** モジュールから **ConfigSource** を使用する場合は、**<module name="org.eclipse.microprofile.config.api"/>** 依存関係を **path/to/org/example/main/module.xml** ファイルに追加します。

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

このコマンドを実行すると、**microprofile-config-smallrye** サブシステムに以下の XML 設定が指定されます。

```
<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>
```

カスタムの **org.eclipse.microprofile.config.spi.ConfigSource** 実装クラスによって提供されるプロパティーはすべての JBoss EAP デプロイメントで使用できます。

3.2.4. ConfigSourceProvider クラスからの ConfigSource 設定の取得

複数の **ConfigSource** インスタンスの実装を登録する、カスタムの **org.eclipse.microprofile.config.spi.ConfigSourceProvider** 実装クラスを作成および設定できます。

手順

- **config-source-provider** を作成します。

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

このコマンドは、**org.example** という名前の JBoss Module によって提供される、**org.example.MyConfigSourceProvider** という名前の実装クラスの **config-source-provider** を作成します。

org.example モジュールから **config-source-provider** を使用する場合は、**<module name="org.eclipse.microprofile.config.api"/>** 依存関係を **path/to/org/example/main/module.xml** ファイルに追加します。

このコマンドを実行すると、**microprofile-config-smallrye** サブシステムに以下の XML 設定が指定されます。

```
<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 実装によって提供されるプロパティーはすべての JBoss EAP デプロイメントで使用できます。

関連情報

- JBoss EAP サーバーにグローバルモジュールを追加する方法は、JBoss EAP**設定ガイド**の グローバルモジュールの定義 を参照してください。

3.3. MICROPROFILE FAULT TOLERANCE 設定

3.3.1. MicroProfile Fault Tolerance 拡張の追加

MicroProfile Fault Tolerance 拡張は、JBoss EAP XP の一部として提供される **standalone-microprofile.xml** および **standalone-microprofile-ha.xml** 設定に含まれています。

エクステンションは標準の **standalone.xml** 設定に含まれません。エクステンションを使用するには、手動で有効にする必要があります。

前提条件

- EAP XP パックがインストールされている。

手順

1. 以下の管理 CLI コマンドを使用して MicroProfile Fault Tolerance 拡張を追加します。

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

2. 以下の managenent コマンドを使用して、**microprofile-fault-tolerance-smallrye** サブシステムを有効にします。

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

3. 以下の管理コマンドでサーバーをリロードします。

```
reload
```

3.4. MICROPROFILE HEALTH 設定

3.4.1. 管理 CLI を使用した正常性の検証

管理 CLI を使用してシステムの正常性を確認できます。

手順

- 正常性を確認します。

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

3.4.2. 管理コンソールを使用した正常性の検証

管理コンソールを使用してシステムの正常性を確認できます。

チェックランタイム操作では、ヘルスチェックとグローバルの結果がブール値として表示されます。

手順

1. **Runtime** タブに移動し、サーバーを選択します。
2. **Monitor** の列で **MicroProfile Health** → **View** の順にクリックします。

3.4.3. HTTP エンドポイントを使用した正常性の検証

正常性検証は JBoss EAP の正常性コンテキストに自動的にデプロイされるため、HTTP エンドポイントを使用して現在の正常性を取得できます。

管理インターフェイスからアクセスできる **/health** エンドポイントのデフォルトアドレスは **http://127.0.0.1:9990/health** です。

手順

- HTTP エンドポイントを使用して、サーバーの現在のヘルス状態を取得するには、以下の URL を使用します。

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

  このコンテキストにアクセスすると、サーバーの状態を示すヘルスチェックが JSON 形式で表示されます。

3.4.4. MicroProfile Health の認証の有効化

アクセスに認証を要求するように **health** コンテキストを設定できます。

手順

1. **microprofile-health-smallrye** サブシステムで **security-enabled** 属性を **true** に設定します。

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

2. 変更を反映するためにサーバーをリロードします。

```
reload
```

/health エンドポイントにアクセスしようとすると、認証プロンプトがトリガーされるようになります。

3.4.5. サーバーの正常性および準備状態を判断する readiness プローブ

JBoss EAP XP 4.0.0 は、サーバーの正常性と readiness を判断するために 3 つの readiness プローブをサポートします。

- **server-status**: server-state は **running** のとき、**UP** を返します。
- **boot-errors**: プローブがブートエラーを検出しないときに **UP** を返します。
- **deployment-status**: すべてのデプロイメントのステータスが **OK** の場合は **UP** を返します。

これらの readiness プローブはデフォルトで有効にされます。MicroProfile Config プロパティー **mp.health.disable-default-procedures** を使用してプローブを無効にすることができます。

以下の例は、**check** 操作で 3 つのプローブを使用する方法を示しています。

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

```
            },
            {
                "name" => "empty-startup-checks",
                "status" => "UP"
            }
        ]
    }
}
```

関連情報

- MicroProfile Health in JBoss EAP
- プローブが定義されていない場合のグローバルステータス

3.4.6. プローブが定義されていない場合のグローバルステータス

:empty-readiness-checks-status、**:empty-liveness-checks-status**、および **:empty-startup-checks-status** 管理属性は、**readiness**、**liveness**、または **startup** プローブが定義されていない場合のグローバルステータスを指定します。

これらの属性により、アプリケーションは、そのアプリケーションが ready/live または、started up であることをプローブが確認するまで、DOWN を報告できるようになります。デフォルトでは、アプリケーションは 'UP' を報告します。

- **:empty-readiness-checks-status** 属性は、**readiness** プローブが定義されていない場合に、**readiness** プローブのグローバルステータスを指定します。

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

- **:empty-liveness-checks-status** 属性は、**liveness** プローブが定義されていない場合に、**liveliness** プローブのグローバルステータスを指定します。

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

- **:empty-startup-checks-status** 属性は、**startup** プローブが定義されていない場合に、**startup** プローブのグローバルステータスを指定します。

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

/**health** HTTP エンドポイントと、**readiness**、**liveness**、**startup** プローブをチェックする **:check** 操作も、これらの属性を考慮に入れます。

これらの属性は以下の例のように変更することもできます。

```
/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 設定

3.5.1. microprofile-jwt-smallrye サブシステムの有効化

MicroProfile JWT 統合は **microprofile-jwt-smallrye** サブシステムによって提供され、デフォルト設定に含まれています。サブシステムがデフォルト設定に存在しない場合は、以下のように追加できます。

前提条件

- EAP XP がインストールされている。

手順

1. JBoss EAP で MicroProfile JWT smallrye 拡張を有効にします。

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

2. **microprofile-jwt-smallrye** サブシステムを有効にします。

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

3. サーバーをリロードします。

   ```
   reload
   ```

microprofile-jwt-smallrye サブシステムが有効になります。

3.6. MICROPROFILE METRICS 管理

3.6.1. 管理インターフェイスで利用可能なメトリック

JBoss EAP サブシステムメトリックは Prometheus 形式で公開されます。

メトリックは JBoss EAP 管理インターフェイスで自動的に利用できるようになり、以下のコンテキストを使用できます。

- **/metrics/**: MicroProfile 3.0 仕様に指定されたメトリックが含まれます。

- **/metrics/vendor**: メモリープールなどのベンダー固有のメトリックが含まれます。
- **/metrics/application**: MicroProfile Metrics API を使用するデプロイしたアプリケーションおよびサブシステムのメトリックが含まれます。

メトリック名はサブシステムと属性名に基づきます。たとえば、サブシステム **undertow** は、アプリケーションデプロイメントのすべてのサーブレットのメトリック属性 **request-count** を公開します。このメトリックの名前は **jboss_undertow_request_count** です。接頭辞 **jboss** は JBoss EAP をメトリックのソースとして識別します。

3.6.2. HTTP エンドポイントを使用したメトリックの検証

HTTP エンドポイントを使用して JBoss EAP 管理インターフェイスで利用可能なメトリックを確認します。

手順

- curl コマンドを使用します。

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

3.6.3. MicroProfile Metrics HTTP エンドポイントの認証の有効化

ユーザーによるコンテキストのアクセスの承認を要求するように **metrics** コンテキストを設定します。この設定は、**metrics** コンテキストのすべてのサブコンテキストに拡張されます。

手順

1. **microprofile-metrics-smallrye** サブシステムで **security-enabled** 属性を **true** に設定します。

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

2. 変更を反映するためにサーバーをリロードします。

```
reload
```

metrics エンドポイントにアクセスしようとすると、認証プロンプトが表示されるようになります。

3.6.4. Web サービスの要求数の取得

要求カウントメトリックを公開する Web サービスの要求数を取得します。

以下の手順では、リクエスト数を取得するために **helloworld-rs** クイックスタートを Web サービスとして使用します。クイックスタートは jboss-eap-quickstarts からクイックスタートをダウンロードします。

前提条件

- Web サービスが要求数を公開している。

手順

1. **undertow** サブシステムの統計を有効にします。

- 統計が有効な状態でスタンドアロンサーバーを起動します。

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

- すでにサーバーが稼働している場合は、**undertow** サブシステムの統計を有効にします。

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

2. **helloworld-rs** クイックスタートをデプロイします。

   - クイックスタートのルートディレクトリーに、Maven を使用して web アプリケーションをデプロイします。

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

3. **curl** コマンドを使用して CLI で http エンドポイントをクエリーし、 **request_count** に対してフィルター処理を行います。

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

   想定される出力:

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

   返された属性値は **0.0** です。

4. Web ブラウザーで http://localhost:8080/helloworld-rs/ にあるクイックスタートにアクセスし、任意のリンクをクリックします。

5. CLI から HTTP エンドポイントを再度クエリーします。

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

   想定される出力:

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

   値は **1.0** に更新されました。

   最後の 2 つの手順を繰り返して、要求数が更新されていることを確認します。

3.7. MICROPROFILE OPENAPI の管理

3.7.1. MicroProfile OpenAPI の有効化

microprofile-openapi-smallrye サブシステムは、**standalone-microprofile.xml** 設定で提供されます。しかし、JBoss EAP XP はデフォルトで **standalone.xml** を使用します。使用するには、**standalone.xml** にサブシステムを含める必要があります。

または、Updating standalone configurations with MicroProfile subsystems and extensions の手順に従い、**standalone.xml** 設定ファイルを更新できます。

手順

1. JBoss EAP で MicroProfile OpenAPI smallrye 拡張を有効にします。

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

2. 以下の管理コマンドを使用して **microprofile-openapi-smallrye** サブシステムを有効にします。

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

3. サーバーをリロードします。

```
reload
```

microprofile-openapi-smallrye サブシステムが有効化されます。

3.7.2. Accept HTTP ヘッダーを使用した MicroProfile OpenAPI ドキュメントリクエスト

Accept HTTP ヘッダーを使用してデプロイメントから MicroProfile OpenAPI ドキュメントをリクエストします。

デフォルトでは、OpenAPI エンドポイントはで YAML ドキュメントを返します。

前提条件

- クエリーされるデプロイメントは、MicroProfile OpenAPI ドキュメントを返すように設定されます。

手順

- 以下の **curl** コマンドを実行して、デプロイメントの **/openapi** エンドポイントをクエリーします。

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

http://localhost:8080 を、デプロイメントの URL およびポートに置き換えます。

Accept ヘッダーは、JSON ドキュメントが **application/json** 文字列を使用して返されることを示します。

3.7.3. HTTP パラメーターを使用した MicroProfile OpenAPI ドキュメントのリクエスト

HTTP リクエストでクエリーパラメーターを使用してデプロイメントから MicroProfile OpenAPI ドキュメントを JSON 形式でリクエストします。

デフォルトでは、OpenAPI エンドポイントはで YAML ドキュメントを返します。

前提条件

- クエリーされるデプロイメントは、MicroProfile OpenAPI ドキュメントを返すように設定されます。

手順

- 以下の **curl** コマンドを実行して、デプロイメントの **/openapi** エンドポイントをクエリーします。

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

http://localhost:8080 を、デプロイメントの URL およびポートに置き換えます。

HTTP パラメーターの **format=JSON** は JSON ドキュメントが返されることを示します。

3.7.4. 静的 OpenAPI ドキュメントを提供するよう JBoss EAP を設定

ホストの REST サービスを記述する静的 OpenAPI ドキュメントに対応するように JBoss EAP を設定します。

JBoss EAP が静的 OpenAPI ドキュメントを提供するよう設定されている場合、静的 OpenAPI ドキュメントは Jakarta RESTful Web Services および MicroProfile OpenAPI アノテーションの前に処理されます。

実稼働環境では、静的ドキュメントを提供するときにアノテーション処理を無効にします。アノテーション処理を無効にすると、イミュータブルでバージョン付けできない API コントラクトがクライアントで利用可能になります。

手順

1. アプリケーションソースツリーにディレクトリーを作成します。

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

APPLICATION_ROOT は、アプリケーションの **pom.xml** 設定ファイルが含まれるディレクトリーです。

2. OpenAPI エンドポイントをクエリーし、出力をファイルにリダイレクトします。

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

デフォルトでは、エンドポイントは YAML ドキュメントを提供し、**format=JSON** は JSON ドキュメントを返すことを指定します。

3. OpenAPI ドキュメントモデルの処理時にアノテーションのスキャンを省略するようにアプリケーションを設定します。

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

4. アプリケーションをリビルドします。

```
$ mvn clean install
```

5. 以下の管理 CLI コマンドを使用してアプリケーションを再度デプロイします。
   a. アプリケーションのアンデプロイ:

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

   b. アプリケーションのデプロイ:

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

JBoss EAP は OpenAPI エンドポイントで静的 OpenAPI ドキュメントを提供するようになりました。

3.7.5. microprofile-openapi-smallrye の無効化

管理 CLI を使用すると、JBoss EAP XP の **microprofile-openapi-smallrye** サブシステムを無効にすることができます。

手順

- **microprofile-openapi-smallrye** サブシステムを無効にします。

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

3.8. MICROPROFILE REACTIVE MESSAGING の管理

3.8.1. JBoss EAP に必要な MicroProfile リアクティブメッセージングエクステンションとサブシステムの設定

JBoss EAP のインスタンスに対して非同期リアクティブメッセージングを有効にする場合は、JBoss EAP 管理 CLI を介してそのエクステンションを追加する必要があります。

前提条件

- SmallRye エクステンションとサブシステムを備えた Reactive Streams Operators を追加しました。詳細については、MicroProfile Reactive Streams Operators Subsystem Configuration: Required Extension を参照してください。
- SmallRye エクステンションとサブシステムを使用したリ Reactive Messaging を追加しました。

手順

1. JBoss EAP 管理 CLI を開きます。
2. 次のコードを入力します。

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

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

注記

OpenShift であるかどうかにかかわらず、Galleon を使用してサーバーをプロビジョニングする場合は、**microprofile-reactive-messaging** Galleon レイヤーを含めて、コアの MicroProfile 2.0.1 とリアクティブメッセージング機能を取得し、必要なサブシステムとエクステンションを有効にしてください。この設定には、Kafka コネクター機能を有効にするために必要な JBoss EAP モジュールが含まれていないことに注意してください。これを行うには、**microprofile-reactive-messaging-kafka** レイヤーを使用します。

検証

管理 CLI の結果のコードで 2 つの場所で **success** が見られた場合は、JBoss EAP に必要な MicroProfile Reactive Messaging エクステンションとサブシステムが正常に追加されています。

ヒント

結果のコードに **reload-required** と表示されている場合は、サーバー設定をリロードして、すべての変更を完全に適用する必要があります。リロードするには、スタンドアロンサーバー CLI で **reload** と入力します。

3.9. スタンドアロンサーバー設定

3.9.1. スタンドアロンサーバー設定ファイル

JBoss EAP XP に、スタンドアロン設定ファイル **standalone-microprofile.xml** および **standalone-microprofile-ha.xml** が含まれるようになりました。

JBoss EAP に含まれる標準設定ファイルは変更されません。JBoss EAP XP 4.0.0 は **domain.xml** ファイルまたはドメインモードの使用をサポートしていないことに注意してください。

表3.1 JBoss EAP XP で利用可能なスタンドアロン設定ファイル

設定ファイル	目的	含まれる機能	除外された機能
standalone.xml	これは、スタンドアロンサーバーの起動時に使用されるデフォルト設定です。	サブシステム、ネットワーキング、デプロイメント、ソケットバインディング、およびその他の設定詳細など、サーバーに関するすべての情報が含まれます。	メッセージングまたは高可用性に必要なサブシステムを除外します。

設定ファイル	目的	含まれる機能	除外された機能
standalone-microprofile.xml	この設定ファイルは、MicroProfile を使用するアプリケーションをサポートします。	サブシステム、ネットワーキング、デプロイメント、ソケットバインディング、およびその他の設定詳細など、サーバーに関するすべての情報が含まれます。	以下の機能を除外します。 ● Jakarta Enterprise Beans ● Messaging ● Jakarta EE Batch ● Jakarta Server Faces ● Jakarta Enterprise Beans タイマー
standalone-ha.xml		デフォルトのサブシステムが含まれ、高可用性のために **modcluster** および **jgroups** サブシステムを追加します。	メッセージングに必要なサブシステムを除外します。
standalone-microprofile-ha.xml	このスタンドアロンファイルは、MicroProfile を使用するアプリケーションをサポートします。	デフォルトのサブシステムに加えて、高可用性向けの **modcluster** および **jgroups** サブシステムが含まれます。	メッセージングに必要なサブシステムを除外します。
standalone-full.xml		デフォルトのサブシステムに加えて、**messaging-activemq** および **iiop-openjdk** サブシステムが含まれます。	
standalone-full-ha.xml	考えられるすべてのサブシステムのサポート。	デフォルトのサブシステムに加えて、メッセージングおよび高可用性のサブシステムが含まれます。	
standalone-load-balancer.xml	ビルトインの mod_cluster フロントエンドロードバランサーを使用して他の JBoss EAP インスタンスの負荷を分散するために必要な最低限のサブシステムのサポート。		

デフォルトでは、スタンドアロンサーバーとして JBoss EAP を起動すると **standalone.xml** ファイルが使用されます。スタンドアロン MicroProfile 設定で JBoss EAP を起動するには、**-c** 引数を使用します。以下に例を示します。

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

関連情報

- JBoss EAP の開始および停止
- 設定データ

3.9.2. MicroProfile サブシステムおよびエクステンションでのスタンドアロン設定の更新

docs/examples/enable-microprofile.cli スクリプトを使用すると、標準のスタンドアロンサーバー設定ファイルを MicroProfile サブシステムおよび拡張機能で更新できます。**enable-microprofile.cli** スクリプトは、カスタム設定ではなく、標準のスタンドアロンサーバー設定ファイルを更新するサンプルスクリプトです。

enable-microprofile.cli スクリプトは、既存のスタンドアロンサーバー設定を変更し、以下の MicroProfile サブシステムおよび拡張機能がない場合はスタンドアロン設定ファイルに追加します。

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

enable-microprofile.cli スクリプトは、変更のハイレベルな説明を出力します。設定は **elytron** サブシステムを使用してセキュア化されます。**security** がある場合は、設定から削除されます。

前提条件

- JBoss EAP XP がインストールされている。

手順

1. 以下の CLI スクリプトを実行して、デフォルトの **standalone.xml** サーバー設定ファイルを更新します。

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

2. 以下のコマンドを使用して、デフォルトの **standalone.xml** サーバー設定ファイル以外のスタンドアロンサーバー設定を選択します。

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

3. 指定した設定ファイルに MicroProfile サブシステムおよび拡張機能が含まれるようになりました。

第4章 JBOSS EAP の MICROPROFILE アプリケーションの開発

4.1. MAVEN および JBOSS EAP MICROPROFILE MAVEN リポジトリー

4.1.1. アーカイブファイルとしての JBoss EAP MicroProfile Maven リポジトリーパッチのダウンロード

MicroProfile Expansion Pack が JBoss EAP に対してリリースされるたびに、JBoss EAP MicroProfile Maven リポジトリーに対応するパッチが提供されます。このパッチは、既存の Red Hat JBoss Enterprise Application Platform 7.4.0 GA Maven リポジトリーに抽出される増分アーカイブファイルとして提供されます。増分アーカイブファイルは既存のファイルを上書きまたは削除しないため、ロールバックの要件はありません。

前提条件

- Red Hat カスタマーポータル でアカウントを設定している。

手順

1. ブラウザーを開き、Red Hat カスタマーポータル にログインします。
2. ページの上部にあるメニューから **Downloads** を選択します。
3. リストで **Red Hat JBoss Enterprise Application Platform** エントリーを見つけ、選択します。
4. **Product** ドロップダウンリストから、**JBoss EAP XP** を選択します。
5. **Version** ドロップダウンリストから **4.0.0** を選択します。
6. **Release** タブをクリックします。
7. リストで **JBoss EAP XP 4.0.0 Incremental Maven Repository**を見つけ、**Download** をクリックします。
8. アーカイブファイルをローカルディレクトリーに保存します。

関連情報

- JBoss EAP Maven リポジトリーの詳細は、JBoss EAP**開発ガイド**の Maven リポジトリー を参照してください。

4.1.2. ローカルシステム上での JBoss EAP MicroProfile Maven リポジトリーパッチの適用

ローカルファイルシステムに JBoss EAP MicroProfile Maven リポジトリーパッチをインストールできます。

増分アーカイブファイルの形式でパッチをリポジトリーに適用すると、新しいファイルがこのリポジトリーに追加されます。増分アーカイブファイルはレポジトリーの既存のファイルを上書きまたは削除しないため、ロールバックの要件はありません。

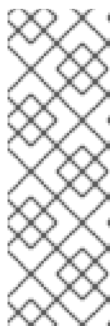
前提条件

- Red Hat JBoss Enterprise Application Platform 7.4.0 GA Maven レポジトリーを ダウンロードし、ローカルシステムにインストール している。
  - ローカルシステムにこのマイナーバージョンの Red Hat JBoss Enterprise Application Platform 7.4 Maven リポジトリーがインストールされていることを確認する。
- ローカルシステムに JBoss EAP XP 4.0.0 Incremental Maven リポジトリーをダウンロードしている。

手順

1. Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven リポジトリーへのパスを見つけます。例: **/path/to/repo/jboss-eap-7.4.0.GA-maven-repository/maven-repository/**
2. ダウンロードした JBoss EAP XP 4.0.0 Incremental Maven リポジトリーを直接 Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven リポジトリーのディレクトリーにデプロイメントします。たとえば、ターミナルを開いて以下のコマンドを実行し、Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven リポジトリーパスの値を置き換えます。

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



注記

EAP_MAVEN_REPOSITORY_PATH は **jboss-eap-7.4.0.GA-maven-repository** を参照します。たとえば、この手順は、**/path/to/repo/jboss-eap-7.4.0.GA-maven-repository/** パスの使用を示しています。

JBoss EAP XP Incremental Maven リポジトリーを Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven リポジトリーに抽出した後、リポジトリー名は JBoss EAP MicroProfile Maven リポジトリーになります。

関連情報

- JBoss EAP Maven リポジトリーの URL を確認するには、JBoss EAP**開発ガイド**の Determining the URL for the JBoss EAP Maven repository を参照してください。

4.1.3. サポートされる JBoss EAP MicroProfile BOM

JBoss EAP XP 4.0.0 には JBoss EAP MicroProfile BOM が含まれています。この BOM は **jboss-eap-xp-microprofile** という名前で、ユースケースは JBoss EAP MicroProfile API をサポートします。

表4.1 JBoss EAP MicroProfile BOM

BOM アーティファクト ID	ユースケース
jboss-eap-xp-microprofile	**groupId** が **org.jboss.bom** のこの BOM は、多くの JBoss EAP MicroProfile がサポートする API 依存関係 (**microprofile-openapi-api** および **microprofile-config-api** など) をパッケージ化します。この BOM を使用する場合は、**jboss-eap-xp-microprofile** BOM が依存関係の値を指定するため、対応の API 依存関係のバージョンを指定する必要はありません。

4.1.4. JBoss EAP MicroProfile Maven リポジトリーの使用

Red Hat JBoss Enterprise Application Platform 7.4.0.GA Maven リポジトリーをインストールし、JBoss EAP XP Incremental Maven リポジトリーを適用した後に **jboss-eap-xp-microprofile** BOM にアクセスできます。その後、リポジトリー名は JBoss EAP MicroProfile Maven リポジトリーになります。BOM は JBoss EAP XP Incremental Maven リポジトリーに同梱されます。

JBoss EAP MicroProfile Maven リポジトリーを使用するには、以下のいずれかを設定する必要があります。

- Maven グローバルまたはユーザー設定
- プロジェクトの POM ファイル

リポジトリーマネージャーや共有サーバー上のリポジトリーを使用して Maven を設定すると、プロジェクトの制御および管理を行いやすくなります。

代替のミラーを使用してプロジェクトファイルを変更せずにリポジトリーマネージャーに特定のリポジトリーのルックアップ要求をすべてリダイレクトすることも可能になります。

警告

POM ファイルを変更して JBoss EAP MicroProfile Maven リポジトリーを設定すると、設定されたプロジェクトのグローバルおよびユーザー Maven 設定が上書きされます。

前提条件

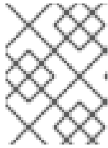
- ローカルシステムに Red Hat JBoss Enterprise Application Platform 7.4 Maven リポジトリーをインストールし、JBoss EAP XP Incremental Maven リポジトリーを適用している。

手順

1. 設定方法を選択し、JBoss EAP MicroProfile Maven リポジトリーを設定します。
2. JBoss EAP MicroProfile Maven リポジトリーを設定したら、**jboss-eap-xp-microprofile** BOM をプロジェクトの POM ファイルに追加します。以下の例は、**pom.xml** ファイルの **<dependencyManagement>** セクションで BOM を設定する方法を示しています。

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

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



注記

pom.xml ファイルに **type** 要素の値を指定しない場合、Maven は要素に **jar** 値を指定します。

関連情報

- JBoss EAP Maven リポジトリーの設定方法の選択に関する詳細は、JBoss EAP **開発ガイド**の Maven リポジトリーの使用 を参照してください。
- 依存関係の管理の詳細は、依存関係管理 を参照してください。

4.2. MICROPROFILE CONFIG の開発

4.2.1. MicroProfile Config の Maven プロジェクトの作成

必要な依存関係で Maven プロジェクトを作成し、MicroProfile Config アプリケーションを作成するためのディレクトリー構造を作成します。

前提条件

- Maven がインストールされている。

手順

1. Maven プロジェクトを設定します。

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

これにより、プロジェクトのディレクトリー構造と **pom.xml** 設定ファイルが作成されます。

2. POM ファイルが **jboss-eap-xp-microprofile** BOM の MicroProfile Config アーティファクトおよび MicroProfile REST Client アーティファクトのバージョンを自動的に管理できるようにするには、POM ファイルの **<dependencyManagement>** セクションに BOM をインポートします。

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

```
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

3. BOM によって管理される MicroProfile Config アーティファクトおよび MicroProfile REST Client アーティファクトおよびその他依存関係をプロジェクト POM ファイルの **<dependency>** セクションに追加します。以下の例は、MicroProfile Config および MicroProfile REST Client 依存関係をファイルに追加する方法を示しています。

```
<!-- 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. アプリケーションでの MicroProfile Config プロパティーの使用

設定された **ConfigSource** を使用するアプリケーションを作成します。

前提条件

- JBoss EAP では MicroProfile Config が有効になります。
- 最新の POM がインストールされている。
- Maven プロジェクトは、MicroProfile Config アプリケーションを作成するために設定されます。

手順

1. クラスファイルを保存するディレクトリーを作成します。

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

APPLICATION_ROOT は、アプリケーションの **pom.xml** 設定ファイルが含まれるディレクトリーです。

2. 新しいディレクトリーに移動します。

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

このディレクトリーに、この手順で説明しているすべてのクラスファイルを作成します。

3. 以下の内容でクラスファイル **HelloApplication.java** を作成します。

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

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

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

}
```

このクラスは、アプリケーションを Jakarta RESTful Web Services アプリケーションとして定義します。

4. 以下の内容を含むクラスファイル **HelloService.java** を作成します。

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

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

5. 以下の内容を含むクラスファイル **HelloWorld.java** を作成します。

```
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 プロパティーは、**@ConfigProperty(name="name", defaultValue="jim")** アノテーションでクラスにインジェクトされます。**ConfigSource** が設定されていない場合、この値 **jim** が返されます。

6. **src/main/webapp/WEB-INF/** ディレクトリーに **beans.xml** という名前の空のファイルを作成します。

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

APPLICATION_ROOT は、アプリケーションの **pom.xml** 設定ファイルが含まれるディレクトリーです。

7. アプリケーションの root ディレクトリーに移動します。

```
$ cd APPLICATION_ROOT
```

APPLICATION_ROOT は、アプリケーションの **pom.xml** 設定ファイルが含まれるディレクトリーです。

8. プロジェクトをビルドします。

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

9. 出力をテストします。

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

以下が想定される出力です。

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

4.3. MICROPROFILE FAULT TOLERANCE アプリケーションの開発

4.3.1. MicroProfile Fault Tolerance 拡張の追加

MicroProfile Fault Tolerance 拡張は、JBoss EAP XP の一部として提供される **standalone-microprofile.xml** および **standalone-microprofile-ha.xml** 設定に含まれています。

エクステンションは標準の **standalone.xml** 設定に含まれません。エクステンションを使用するには、手動で有効にする必要があります。

前提条件

- EAP XP パックがインストールされている。

手順

1. 以下の管理 CLI コマンドを使用して MicroProfile Fault Tolerance 拡張を追加します。

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

2. 以下の managenent コマンドを使用して、**microprofile-fault-tolerance-smallrye** サブシステムを有効にします。

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

3. 以下の管理コマンドでサーバーをリロードします。

```
reload
```

4.3.2. MicroProfile Fault 容認のための Maven プロジェクトの設定

必要な依存関係で Maven プロジェクトを作成し、MicroProfile Fault Tolerance アプリケーションを作成するためのディレクトリー構造を作成します。

前提条件

- Maven がインストールされている。

手順

1. Maven プロジェクトを設定します。

```
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
```

このコマンドは、プロジェクトのディレクトリー構造と **pom.xml** 設定ファイルを作成します。

2. POM ファイルが **jboss-eap-xp-microprofile** BOM の MicroProfile Fault Tolerance アーティファクトのバージョンを自動的に管理できるようにするには、POM ファイルの **<dependencyManagement>** セクションに BOM をインポートします。

```
<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>
```

${version.microprofile.bom} を、インストールされた BOM のバージョンに置き換えます。

3. BOM によって管理される MicroProfile Fault Tolerance アーティファクトをプロジェクト POM ファイルの **<dependency>** セクションに追加します。以下の例は、MicroProfile Fault Tolerance 依存関係をファイルに追加する方法を示しています。

```
<!-- 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. フォールトトレランスアプリケーションの作成

フォールトトレランスを確保するために再試行、タイムアウト、フォールバックパターンを実装するフォールトトレランスアプリケーションを作成します。

前提条件

- Maven 依存関係が設定されている。

手順

1. クラスファイルを保存するディレクトリーを作成します。

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

APPLICATION_ROOT は、アプリケーションの **pom.xml** 設定ファイルが含まれるディレクトリーです。

2. 新しいディレクトリーに移動します。

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

以下の手順では、新しいディレクトリーにすべてのクラスファイルを作成します。

3. 以下の内容で、**Coffee.java** としてクロージサンプルを表す単純なエンティティーを作成します。

```
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. 以下の内容でクラスファイル **CoffeeApplication.java** を作成します。

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

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

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

5. 以下の内容で Jakarta Contexts and Dependency Injection Bean を **CoffeeRepositoryService.java** として作成します。

```
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. 以下の内容でクラスファイル **CoffeeResource.java** を作成します。

```
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. **4** への再試行回数を定義します。
2. タイムアウト間隔をミリ秒単位で定義します。
3. 呼び出しに失敗した場合に呼び出されるフォールバックメソッドを定義します。

7. アプリケーションの root ディレクトリーに移動します。

```
$ cd APPLICATION_ROOT
```

8. 以下の Maven コマンドを使用してアプリケーションをビルドします。

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

http://localhost:8080/microprofile-fault-tolerance/coffee でアプリケーションにアクセスします。

関連情報

- アプリケーションの耐障害性をテストするためのエラーを含むフォールトトレランスアプリケーションの詳細は、**microprofile-fault-tolerance** クイックスタートを参照してください。

4.4. MICROPROFILE HEALTH の開発

4.4.1. カスタムヘルスチェックの例

microprofile-health-smallrye サブシステムによって提供されるデフォルトの実装は基本的なヘルスチェックを実行します。サーバーやアプリケーションの状態の詳細情報はカスタムヘルスチェックに含まれる可能性があります。クラスレベルで **org.eclipse.microprofile.health.Liveness**、**org.eclipse.microprofile.health.Readiness**、または **org.eclipse.microprofile.health.Startup** アノテーションを含む Jakarta コンテキスト and Dependency Injection は、実行時に自動的に検出されて呼び出されます。

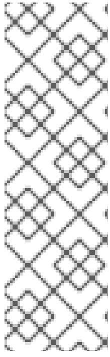
以下の例は、**UP** 状態を返すヘルスチェックの新しい実装を作成する方法を表しています。

```
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();
    }
}
```

次の例に示すように、ヘルスチェックをデプロイした後、後続のヘルスチェッククエリーにはカスタムチェックが含まれます。

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



注記

次のコマンドを使用して、liveness、readiness、および startup のチェックを行うことができます。

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

4.4.2. @Liveness アノテーションの例

次の例は、アプリケーションで **@Liveness** アノテーションを使用する方法を示しています。

```
@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 アノテーションの例

次の例は、データベースへの接続を確認する方法を示しています。データベースがダウンしている場合は、readiness チェックでエラーが報告されます。

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

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

    @Override
    public HealthCheckResponse call() {

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

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

```
            }

            return responseBuilder.build();
        }

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

4.4.4. @Startup アノテーションの例

以下は、アプリケーションで **@Startup** アノテーションを使用する例です。

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

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

4.5. MICROPROFILE JWT アプリケーションの開発

4.5.1. microprofile-jwt-smallrye サブシステムの有効化

MicroProfile JWT 統合は **microprofile-jwt-smallrye** サブシステムによって提供され、デフォルト設定に含まれています。サブシステムがデフォルト設定に存在しない場合は、以下のように追加できます。

前提条件

- EAP XP がインストールされている。

手順

1. JBoss EAP で MicroProfile JWT smallrye 拡張を有効にします。

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

2. **microprofile-jwt-smallrye** サブシステムを有効にします。

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

3. サーバーをリロードします。

```
reload
```

microprofile-jwt-smallrye サブシステムが有効になります。

4.5.2. JWT アプリケーションを開発するための Maven プロジェクトの設定

必要な依存関係と JWT アプリケーションを開発するためのディレクトリー構造で Maven プロジェクトを作成します。

前提条件

- Maven がインストールされている。
- **microprofile-jwt-smallrye** サブシステムが有効になっている。

手順

1. Maven プロジェクトを設定します。

```
$ 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
```

このコマンドは、プロジェクトのディレクトリー構造と **pom.xml** 設定ファイルを作成します。

2. POM ファイルが **jboss-eap-xp-microprofile** BOM の MicroProfile JWT アーティファクトのバージョンを自動的に管理できるようにするには、POM ファイルの **<dependencyManagement>** セクションに BOM をインポートします。

```
<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>
```

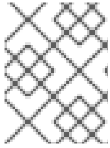
${version.microprofile.bom} を、インストールされた BOM のバージョンに置き換えます。

3. BOM によって管理される MicroProfile JWT アーティファクトをプロジェクト POM ファイルの **<dependency>** セクションに追加します。以下の例は、MicroProfile JWT 依存関係をファイルに追加する方法を示しています。

```
<!-- 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. MicroProfile JWT を使用したアプリケーションの作成

JWT トークンに基づいてリクエストを認証し、トークンベアラーのアイデンティティーに基づいて承認を実装するアプリケーションを作成します。



注記

以下の手順では、トークンを生成するサンプルコードを説明します。実稼働環境では、Red Hat Single Sign-on (SSO) などのアイデンティティープロバイダーを使用します。

前提条件

- Maven プロジェクトが正しい依存関係で設定されている。

手順

1. トークンジェネレーターを作成します。
   この手順は参照用です。実稼働環境では、Red Hat SSO などのアイデンティティープロバイダーを使用します。
   a. トークンジェネレーターユーティリティーの **src/test/java** ディレクトリーを作成し、これに移動します。

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

   b. 以下の内容でクラスファイル **TokenUtil.java** を作成します。

   ```
   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. 以下の内容を含む **src/main/webapp/WEB-INF** ディレクトリーに **web.xml** ファイルを作成します。

```
<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. 以下の内容でクラスファイル **SampleEndPoint.java** を作成します。

```
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.";

    }

   }
```

@Path アノテーション付きの メソッドは Jakarta RESTful Web Services エンドポイントです。

@Claim アノテーションは JWT 要求を定義します。

4. クラスファイル **App.java** を作成して Jakarta RESTful Web Services を有効にします。

```
   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 {}
```

アノテーション **@LoginConfig(authMethod="MP-JWT", realmName="MP JWT Realm")** は、デプロイメント中に JWT RBAC を有効にします。

5. 以下の Maven コマンドを使用してアプリケーションをコンパイルします。

```
$ mvn package
```

6. トークンジェネレーターユーティリティーを使用して JWT トークンを生成します。

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

7. 以下の Maven コマンドを使用してアプリケーションをビルドおよびデプロイします。

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

8. アプリケーションをテストします。

   - ベアラートークンを使用して **Sample/subscription** エンドポイントを呼び出します。

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

   - **Sample/birthday** エンドポイントを呼び出します。

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

4.6. MICROPROFILE METRICS の開発

4.6.1. MicroProfile Metrics アプリケーションの作成

アプリケーションに対して行われるリクエスト数を返すアプリケーションを作成します。

手順

1. 以下の内容を含むクラスファイル **HelloService.java** を作成します。

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

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

2. 以下の内容を含むクラスファイル **HelloWorld.java** を作成します。

```
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. 以下の依存関係を含めるように **pom.xml** ファイルを更新します。

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

4. 以下の Maven コマンドを使用してアプリケーションをビルドします。

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

5. メトリックをテストします。

   a. CLI で以下のコマンドを実行します。

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

   想定される出力:

   ```
   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. ブラウザーで http://localhost:8080/helloworld-rs/rest/json にアクセスします。

   c. CLI で以下のコマンドを再度実行します。

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

想定される出力:

```
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. MICROPROFILE OPENAPI アプリケーションの開発

4.7.1. MicroProfile OpenAPI の有効化

microprofile-openapi-smallrye サブシステムは、**standalone-microprofile.xml** 設定で提供されます。しかし、JBoss EAP XP はデフォルトで **standalone.xml** を使用します。使用するには、**standalone.xml** にサブシステムを含める必要があります。

または、Updating standalone configurations with MicroProfile subsystems and extensions の手順に従い、**standalone.xml** 設定ファイルを更新できます。

手順

1. JBoss EAP で MicroProfile OpenAPI smallrye 拡張を有効にします。

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

2. 以下の管理コマンドを使用して **microprofile-openapi-smallrye** サブシステムを有効にします。

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

3. サーバーをリロードします。

```
reload
```

microprofile-openapi-smallrye サブシステムが有効化されます。

4.7.2. MicroProfile OpenAPI の Maven プロジェクトの設定

Maven プロジェクトを作成し、MicroProfile OpenAPI アプリケーションを作成するための依存関係を設定します。

前提条件

- Maven がインストールされている。
- JBoss EAP Maven リポジトリーが設定されている。

手順

1. プロジェクトを初期化します。

```
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
```

このコマンドは、プロジェクトのディレクトリー構造と **pom.xml** 設定ファイルを作成します。

2. **pom.xml** 設定ファイルを編集して以下を追加します。

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

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

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

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

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

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

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

  <build>
```

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

pom.xml 設定ファイルおよびディレクトリー構造を使用してアプリケーションを作成します。

関連情報

- JBoss EAP Maven リポジトリーの設定に関する詳細は、POM ファイルを使用した JBoss EAP Maven リポジトリーの設定 を参照してください。

4.7.3. MicroProfile OpenAPI アプリケーションの開発

OpenAPI v3 ドキュメントを返すアプリケーションを作成します。

前提条件

- Maven プロジェクトは、MicroProfile OpenAPI アプリケーションを作成するために設定されます。

手順

1. クラスファイルを保存するディレクトリーを作成します。

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

   APPLICATION_ROOT は、アプリケーションの **pom.xml** 設定ファイルが含まれるディレクトリーです。

2. 新しいディレクトリーに移動します。

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

   以下の手順のクラスファイルすべては、このディレクトリーに作成する必要があります。

3. 以下の内容でクラスファイル **InventoryApplication.java** を作成します。

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

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

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

   このクラスはアプリケーションの REST エンドポイントとして機能します。

4. 以下の内容でクラスファイル **Fruit.java** を作成します。

```
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. 以下の内容でクラスファイル **FruitResource.java** を作成します。

```
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. アプリケーションの root ディレクトリーに移動します。

```
$ cd APPLICATION_ROOT
```

7. 以下の Maven コマンドを使用してアプリケーションをビルドおよびデプロイします。

```
$ mvn wildfly:deploy
```

8. アプリケーションをテストします。

- **curl** を使用して、サンプルアプリケーションの OpenAPI ドキュメントにアクセスします。

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

- 以下の出力が返されます。


```
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
```

関連情報

- MicroProfile SmallRye OpenAPI で定義されたアノテーションのリストは、MicroProfile OpenAPI annotations を参照してください。

4.7.4. 静的 OpenAPI ドキュメントを提供するよう JBoss EAP を設定

ホストの REST サービスを記述する静的 OpenAPI ドキュメントに対応するように JBoss EAP を設定します。

JBoss EAP が静的 OpenAPI ドキュメントを提供するよう設定されている場合、静的 OpenAPI ドキュメントは Jakarta RESTful Web Services および MicroProfile OpenAPI アノテーションの前に処理されます。

実稼働環境では、静的ドキュメントを提供するときにアノテーション処理を無効にします。アノテーション処理を無効にすると、イミュータブルでバージョン付けできない API コントラクトがクライアントで利用可能になります。

手順

1. アプリケーションソースツリーにディレクトリーを作成します。

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

   APPLICATION_ROOT は、アプリケーションの **pom.xml** 設定ファイルが含まれるディレクトリーです。

2. OpenAPI エンドポイントをクエリーし、出力をファイルにリダイレクトします。

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

   デフォルトでは、エンドポイントは YAML ドキュメントを提供し、**format=JSON** は JSON ドキュメントを返すことを指定します。

3. OpenAPI ドキュメントモデルの処理時にアノテーションのスキャンを省略するようにアプリケーションを設定します。

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

4. アプリケーションをリビルドします。

   ```
   $ mvn clean install
   ```

5. 以下の管理 CLI コマンドを使用してアプリケーションを再度デプロイします。

a. アプリケーションのアンデプロイ:

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

b. アプリケーションのデプロイ:

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

JBoss EAP は OpenAPI エンドポイントで静的 OpenAPI ドキュメントを提供するようになりました。

4.8. MICROPROFILE REST クライアントの開発

4.8.1. MicroProfile REST クライアントおよび Jakarta RESTful Web Services 構文の比較

MicroProfile REST クライアントは、CORBA、Java Remote Method Invocation(RMI)、JBoss Remoting Project、RESTEasy にも実装される分散オブジェクト通信のバージョンを有効にします。たとえば、リソースについて考えてみましょう。

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

以下の例は、Jakarta RESTful Web Services ネイティブ方法を使用して **TestResource** クラスにアクセスする方法を示しています。

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

ただし、Microprofile REST クライアントは、以下の例のように **test()** メソッドを直接呼び出すことで、より直感的な構文をサポートします。

```
@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();
```

上記の例では、**TestResource** クラスでの呼び出しは、**service.test()** の呼び出しにあるように **TestResourceIntf** クラスを使用すると大幅に容易になります。

以下の例は、**TestResourceIntf** のより詳細なバージョンです。

```
@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);
}
```

service.test("p", "q", "e") メソッドを呼び出すと、以下の例のように HTTP メッセージが表示されます。

```
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 クライアントでのプロバイダーのプログラムによる登録

MicroProfile REST クライアントを使用して、プロバイダーを登録してクライアント環境を設定できます。以下に例を示します。

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

4.8.3. MicroProfile REST クライアントでのプロバイダーの宣言的登録

以下の例のように **org.eclipse.microprofile.rest.client.annotation.RegisterProvider** アノテーションをターゲットインターフェイスに追加すると、MicroProfile REST クライアントを 使用してプロバイダーを宣言で登録します。

```
@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);
}
```

MyClientResponseFilter クラスと **MyMessageBodyReader** クラスをアノテーションで宣言すると、**RestClientBuilder.register()** メソッドを呼び出す必要がなくなります。

4.8.4. MicroProfile REST クライアントでのヘッダーの宣言型仕様

HTTP リクエストのヘッダーは、以下の方法で指定できます。

- リソースメソッドパラメーターのいずれかにアノテーションを付けます。
- **org.eclipse.microprofile.rest.client.annotation.ClientHeaderParam** アノテーションを宣言で使用。

以下の例では、**@HeaderParam** アノテーションを持つリソースメソッドパラメーターのいずれかにアノテーションを付け、ヘッダーの設定を示しています。

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

以下の例は、**org.eclipse.microprofile.rest.client.annotation.ClientHeaderParam** アノテーションを使用してヘッダーを設定する例になります。

```
@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 クライアントの ResponseExceptionMapper

org.eclipse.microprofile.rest.client.ext.ResponseExceptionMapper クラスは、Jakarta RESTful Web Services で定義される **javax.ws.rs.ext.ExceptionMapper** クラスとは逆のクライアント側です。**ExceptionMapper.toResponse()** メソッドは、サーバー側の処理中に発生する **Exception** クラスを **Response** クラスに変換します。**ResponseExceptionMapper.toThrowable()** メソッドは、HTTP エラーステータスでクライアント側で受信した **Response** クラスを **Exception** クラスに変換します。

ResponseExceptionMapper クラスは、プログラムまたは宣言で登録できます。登録された **ResponseExceptionMapper** クラスがない場合、デフォルトの **ResponseExceptionMapper** クラスはステータス **>= 400** のレスポンスを **WebApplicationException** クラスにマップします。

4.8.6. MicroProfile REST クライアントでのコンテキスト依存関係の挿入

MicroProfile REST クライアントでは、**@RegisterRestClient** クラスで、Jakarta Contexts and Dependency Injection(Jakarta Contexts and Dependency Injection)Bean として管理されるインターフェイスにアノテーションを付ける必要があります。以下に例を示します。

```
@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);
  }
```

ここで、MicroProfile REST クライアント実装は **TestDataBase** クラスサービスのクライアントを作成し、**TestResourceImpl** クラスによるアクセスを容易にします。ただし、**TestDataBase** クラス実装へのパスに関する情報は含まれません。この情報は、オプションの **@RegisterProvider** パラメーター **baseUri** で指定できます。

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

これは、https://localhost:8080/webapp で **TestDataBase** の実装にアクセスできることを示しています。MicroProfile 設定を使用して情報を外部で提供することもできます。

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

たとえば、以下のコマンドは、https://localhost:8080/webapp にある **com.bluemonkeydiamond.TestDatabase** クラスの実装にアクセスできることを示しています。

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

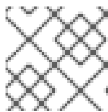
Jakarta Contexts and Dependency Injection クライアントに他のプロパティーを多数指定できます。たとえば、**com.mycompany.remoteServices.MyServiceClient/mp-rest/providers** と、クライアントに含める完全修飾プロバイダークラス名のコンマ区切りリスト。

関連情報

- MicroProfile REST Client 仕様の詳細は、MicroProfile の Rest Client を参照してください。
- MicroProfile REST Client 2.0 機能の詳細は、MicroProfile REST Client 2.0 を参照してください。

第5章 JBOSS EAP XP の OPENSHIFT イメージでマイクロサービスアプリケーションをビルドおよび実行

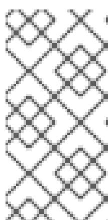
JBoss EAP XP の OpenShift イメージでマイクロサービスアプリケーションをビルドし、実行できます。



注記

JBoss EAP XP は、OpenShift 4 以降のバージョンでのみサポートされます。

以下のワークフローを使用して、Source-to-image (S2I) プロセスで JBoss EAP XP の OpenShift イメージでマイクロサービスアプリケーションをビルドし、実行します。



注記

JBoss EAP XP 4.0.0 の OpenShift イメージは、**standalone-microprofile-ha.xml** ファイルをベースとしたデフォルトのスタンドアロン設定ファイルを提供します。JBoss EAP XP に含まれるサーバー設定ファイルの詳細は、**スタンドアロンサーバー設定ファイル**を参照してください。

このワークフローでは、例として **microprofile-config** クイックスタートを使用します。クイックスタートでは、独自のプロジェクトの参照として使用できる小規模の、特定の作業例を示します。詳細は、JBoss EAP XP 4.0.0 に同梱される **microprofile-config** クイックスタートを参照してください。

関連情報

- JBoss EAP XP に含まれるサーバー設定ファイルの詳細は、スタンドアロンサーバー設定ファイル を参照してください。

5.1. アプリケーションのデプロイメントに向けた OPENSHIFT の準備

アプリケーションのデプロイメントに向けて OpenShift を準備します。

前提条件

稼働中の OpenShift インスタンスがインストールされている。詳細は、Red Hat カスタマーポータルの**OpenShift Container Platform クラスターのインストールおよび設定**を参照してください。

手順

1. **oc login** コマンドを使用して、OpenShift インスタンスにログインします。
2. OpenShift で新しいプロジェクトを作成します。
   プロジェクトでは、1 つのユーザーグループが他のグループとは別にコンテンツを整理および管理することができます。以下のコマンドを使用すると OpenShift でプロジェクトを作成できます。

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

   たとえば、以下のコマンドを使用して、**microprofile-config** クイックスタートで **eap-demo** という名前の新規プロジェクトを作成します。

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

5.2. RED HAT コンテナーレジストリーへの認証の設定

JBoss EAP XP の OpenShift イメージをインポートおよび使用するには、Red Hat コンテナーレジストリーへの認証を設定する必要があります。

レジストリーサービスアカウントを使用して認証トークンを作成し、Red Hat Container Registry へのアクセスを設定します。認証トークンを使用する場合は、Red Hat アカウントのユーザー名とパスワードを OpenShift 設定に使用したり、保存したりする必要はありません。

手順

1. Red Hat カスタマーポータルの手順にしたがって、レジストリーサービスアカウント管理アプリケーション を使用して認証トークンを作成します。
2. トークンの OpenShift シークレットが含まれる YAML ファイルをダウンロードします。
   YAML ファイルは、トークンの **Token Information** ページの **OpenShift Secret** タブからダウンロードできます。
3. ダウンロードした YAML ファイルを使用して、OpenShift プロジェクトの認証トークンシークレットを作成します。

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

4. 以下のコマンドを使用して、OpenShift プロジェクトのシークレットを設定します。シークレット名は前のステップで作成したシークレットの名前に置き換えてください。

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

関連情報

- Red Hat コンテナーレジストリーへの認証の設定
- レジストリーサービスアカウント管理アプリケーション
- 安全なレジストリーにアクセスの設定

5.3. JBOSS EAP XP の最新の OPENSHIFT イメージストリームおよびテンプレートのインポート

JBoss EAP XP の最新の OpenShift イメージストリームおよびテンプレートをインポートします。

重要

OpenShift 上の OpenJDK 8 イメージおよびイメージストリームは非推奨となりました。

イメージおよびイメージストリームは OpenShift でも引き続きサポートされます。ただし、これらのイメージおよびイメージストリームの拡張機能はなく、今後削除される可能性があります。Red Hat は、標準のサポート条件下で、OpenJDK 8 イメージおよびイメージストリームに対するフルサポートおよびバグ修正を継続して提供します。

手順

1. JBoss EAP XP の OpenShift イメージの最新のイメージストリームおよびテンプレートを OpenShift プロジェクトの名前空間にインポートするには、以下のコマンドを使用します。

   a. JDK 11 イメージストリームのインポート:

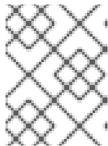
   ```
   oc replace --force -f https://raw.githubusercontent.com/jboss-container-images/jboss-eap-openshift-templates/eap-xp4/eap-xp4-openjdk11-image-stream.json
   ```

   このコマンドは以下のイメージストリームおよびテンプレートをインポートします。

   - JDK 11 ビルダーイメージストリーム: jboss-eap-xp4-openjdk11-openshift
   - JDK 11 ランタイムイメージストリーム: jboss-eap-xp4-openjdk11-runtime-openshift

   b. OpenShift テンプレートをインポートします。

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



注記

上記のコマンドを使用してインポートされた JBoss EAP XP イメージストリームおよびテンプレートは、OpenShift プロジェクト内のみで利用できます。

2. 一般的な **openshift** namespace にアクセスできる管理者権限を持っている場合、すべてのプロジェクトがイメージストリームおよびテンプレートにアクセスできるようにするには、コマンドの **oc replace** 行に **-n openshift** を追加します。以下に例を示します。

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

3. イメージストリームとテンプレートを別のプロジェクトにインポートする必要がある場合には、コマンドラインの **oc replace** に **-n PROJECT_NAME** を追加します。以下に例を示します。

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

   cluster-samples-operator を使用する場合は、クラスターサンプルオペレーターの設定についての OpenShift ドキュメントを参照してください。クラスターサンプルオペレーターの設定の詳細は、サンプルオペレーターの設定 を参照してください。

5.4. OPENSHIFT への JBOSS EAP XP SOURCE-TO-IMAGE (S2I) アプリケーションのデプロイ

JBoss EAP source-to-image (S2I) アプリケーションの OpenShift へのデプロイ

要件

- オプション: テンプレートは、多くのテンプレートパラメーターにデフォルト値を指定でき、一部またはすべてのデフォルトをオーバーライドする必要がある場合があります。パラメーターのリストやデフォルト値などのテンプレートの情報を表示するには、コマンド **oc describe template TEMPLATE_NAME** を使用します。

手順

1. JBoss EAP XP イメージと Java アプリケーションのソースコードを使用して、新しい OpenShift アプリケーションを作成します。S2I ビルド用に提供される JBoss EAP XP テンプレートの 1 つを使用します。

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

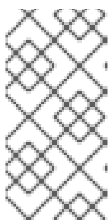
(1) 使用するテンプレート。アプリケーションイメージは **latest** でタグ付けされます。

(2) 最新のイメージストリームとテンプレートは、プロジェクトの namespace にインポートされたため、イメージストリームが見つかる場所の namespace を指定する必要があります。通常はプロジェクトの名前になります。

(3) アプリケーションのソースコードが含まれるリポジトリーの URL。

(4) ソースコードに使用する Git リポジトリー参照。Git ブランチやタグ参照にすることができます。

(5) ビルドするソースリポジトリー内のディレクトリー。



注記

テンプレートは、多くのテンプレートパラメーターにデフォルト値を指定でき、一部またはすべてのデフォルトをオーバーライドする必要がある場合があります。パラメーターのリストやデフォルト値などのテンプレートの情報を表示するには、コマンド **oc describe template TEMPLATE_NAME** を使用します。

新しい OpenShift アプリケーションを作成するときに、環境変数を設定 することもあります。

2. ビルド設定の名前を取得します。

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

3. 取得したビルド設定の名前を使用し、Maven のビルドの進捗を表示します。

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

…
Push successful
```

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

たとえば、**microprofile-config** の場合、以下のコマンドは Maven ビルドの進捗状況を表示します。

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

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

関連情報

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

5.5. JBOSS EAP XP SOURCE-TO-IMAGE (S2I) アプリケーションのデプロイメント後タスクの完了

アプリケーションによっては、OpenShift アプリケーションのビルドおよびデプロイ後に一部のタスクを完了する必要がある場合があります。

デプロイメント後タスクの例には、以下が含まれます。

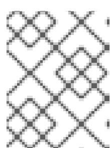
- アプリケーションを OpenShift の外部から表示できるようにサービスを公開します。
- アプリケーションを特定のレプリカ数にスケーリングします。

手順

1. 以下のコマンドを使用してアプリケーションのサービス名を取得します。

```
$ oc get service
```

2. **オプション**: メインサービスをルートとして公開し、OpenShift 外部からアプリケーションにアクセスできるようにします。たとえば、**microprofile-config** クイックスタートでは、以下のコマンドを使用して必要なサービスとポートを公開します。



注記

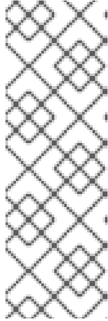
テンプレートを使用してアプリケーションを作成した場合は、ルートがすでに存在することがあります。存在する場合は次のステップに進みます。

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

3. ルートの URL を取得します。

```
$ oc get route
```

4. この URL を使用して web ブラウザーでアプリケーションにアクセスします。URL は前のコマンド出力にある **HOST/PORT** フィールドの値になります。



注記

JBoss EAP XP 4.0.0 GA ディストリビューションでは、Micprofile Config クイックスタートはアプリケーションのルートコンテキストに対して HTTPS GET リクエストに応答しません。今回の機能拡張は、{JBossXPShortName101} GA ディストリビューションでのみ利用可能です。

たとえば、Micprofile Config アプリケーションと対話するには、ブラウザーの URL は **http://HOST_PORT_Value/config/value** になります。

アプリケーションが JBoss EAP ルートコンテキストを使用しない場合、アプリケーションのコンテキストを URL に追加します。たとえば、**microprofile-config** クイックスタートの URL は **http://HOST_PORT_VALUE/microprofile-config/** のようになります。

5. 任意で、以下のコマンドを実行してアプリケーションインスタンスをスケールアップすることもができます。このコマンドにより、レプリカ数が 3 に増えます。

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

たとえば、**microprofile-config** クイックスタートでは、以下のコマンドを使用してアプリケーションをスケールアップします。

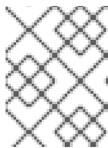
```
$ oc scale deploymentconfig/eap-xp4-basic-app --replicas=3
```

関連情報

JBoss EAP XP クイックスタートの詳細は、JBoss EAP XP クイックスタート を参照してください。

第6章 機能のトリム

起動可能な JAR をビルドする場合、含まれる JBoss EAP の機能とサブシステムを決定できます。



注記

機能のトリムは、OpenShift 上で、または起動可能な JAR のビルド時にのみサポートされます。

関連情報

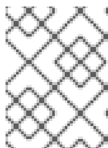
- 起動可能な JAR について

6.1. 利用可能な JBOSS EAP レイヤー

Red Hat は、OpenShift または起動可能な JAR での JBoss EAP サーバーのプロビジョニングをカスタマイズするために複数のレイヤーを提供します。

3 つの層は、コア機能を提供するベースレイヤーです。他のレイヤーは、追加機能を備えたベースレイヤーを強化するデコレーターレイヤーです。

ほとんどのデコレーターレイヤーを使用して、JBoss EAP for OpenShift で S2I イメージをビルドしたり、起動可能な JAR をビルドしたりできます。一部のレイヤーは S2I イメージをサポートしません。レイヤーノートの説明は、この制限の説明になります。



注記

リスト表示されたレイヤーのみがサポートされます。ここで記載されていないレイヤーはサポートされません。

6.1.1. ベースレイヤー

各ベースレイヤーには、典型的なサーバーユーザーケースのコア機能が含まれています。

datasources-web-server
このレイヤーには、サーブレットコンテナーが含まれ、データソースを設定する機能が含まれます。

このレイヤーには MicroProfile 機能が含まれません。

このレイヤーでは、以下の Jakarta EE 仕様がサポートされます。

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

- Jakarta Annotations 1.3
- Jakarta XML Binding 2.3
- Jakarta Debugging Support for Other Languages 1.0
- Jakarta Transaction 1.3
- Jakarta Connector API 1.7

jaxrs-server
このレイヤーは、以下の JBoss EAP サブシステムを使用して **datasources-web-server** レイヤーを強化します。

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

このレイヤーは、コンテナーに Infinispan ベースのセカンドレベルのエンティティーキャッシングをローカルに追加します。

以下の MicroProfile 機能は、このレイヤーに含まれています。

- MicroProfile REST クライアント

以下の Jakarta EE 仕様は、**datasources-web-server** レイヤーでサポートされるものに加え、このレイヤーでサポートされています。

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

cloud-server
このレイヤーは、以下の JBoss EAP サブシステムを使用して **jaxrs-server** レイヤーを強化します。

- **resource-adapters**
- **messaging-activemq** (組み込みメッセージではなく、リモートブラオーカーメッセージング)

このレイヤーは、以下の observability 機能も **jaxrs-server** レイヤーに追加します。

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

以下の Jakarta EE 仕様は、**jaxrs-server** レイヤーでサポートされるものに加え、このレイヤーでサポートされています。

- Jakarta Security 1.0

6.1.2. デコレーターレイヤー

デコレーターレイヤーは単独で使用されません。ベースレイヤーで 1 つ以上のデコレーターレイヤーを設定するとで、追加機能を利用できます。

ejb-lite
このデコレーターレイヤーは、プロビジョニングされたサーバーに最小限の Jakarta Enterprise Bean 実装を追加します。このレイヤーには、以下のサポートは含まれません。

- IIOP の統合
- MDB インスタンスプール
- リモートコネクターリソース

このレイヤーは、起動可能な JAR をビルドする場合のみサポートされます。このレイヤーは、S2I を使用する場合にはサポートされません。

Jakarta Enterprise Beans
このデコレーターレイヤーは、**ejb-lite** レイヤーを拡張します。このレイヤーは、**ejb-lite** レイヤーに含まれるベース機能に加えて、プロビジョニングされたサーバーに以下のサポートを追加します。

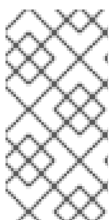
- MDB インスタンスプール
- リモートコネクターリソース

メッセージ駆動 Bean (MDB) または Jakarta Enterprise Beans リモーティング機能の両方を使用する場合は、このレイヤーを使用します。これらの機能が必要ない場合は、**ejb-lite** レイヤーを使用します。

このレイヤーは、起動可能な JAR をビルドする場合のみサポートされます。このレイヤーは、S2I を使用する場合にはサポートされません。

ejb-local-cache
このデコレーターレイヤーは、Jakarta Enterprise Beans のローカルキャッシュサポートをプロビジョニングされたサーバーに追加します。

依存関係: **ejb-lite** レイヤーまたは **ejb** レイヤーを含む場合に限り、このレイヤーを含めることができます。



注記

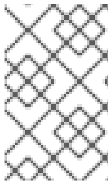
このレイヤーは **ejb-dist-cache** レイヤーと互換性がありません。**ejb-dist-cache** レイヤーが含まれている場合は、**ejb-local-cache** レイヤーを含めることができません。両方のレイヤーが含まれる場合、生成されるビルドには予期しない Jakarta Enterprise Beans 設定が含まれる可能性があります。

このレイヤーは、起動可能な JAR をビルドする場合のみサポートされます。このレイヤーは、S2I を使用する場合にはサポートされません。

ejb-dist-cache

このデコレーターレイヤーは、Jakarta Enterprise Beans の分散キャッシングサポートをプロビジョニングされたサーバーに追加します。

依存関係: **ejb-lite** レイヤーまたは **ejb** レイヤーを含む場合に限り、このレイヤーを含めることができます。



注記

このレイヤーは **ejb-local-cache** レイヤーと互換性がありません。**ejb-dist-cache** レイヤーが含まれている場合は、**ejb-local-cache** レイヤーを含めることができません。両方のレイヤーを含めると、ビルドが予期しない設定になる可能性があります。

このレイヤーは、起動可能な JAR をビルドする場合のみサポートされます。このレイヤーは、S2I を使用する場合にはサポートされません。

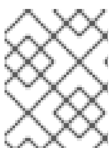
jdr
このデコレーターレイヤーは、Red Hat からサポートを要求する際に診断データを収集するために JBoss Diagnostic Reporting (**jdr**) サブシステムを追加します。

このレイヤーは、起動可能な JAR をビルドする場合のみサポートされます。このレイヤーは、S2I を使用する場合にはサポートされません。

Jakarta Persistence
このデコレーターレイヤーは、単一ノードサーバーの永続性機能を追加します。分散キャッシングは、サーバーがクラスターを形成できる場合にのみ機能することに注意してください。

レイヤーは Hibernate ライブラリーをプロビジョニングされたサーバーに追加し、以下のサポートを受けることができます。

- **jpa** サブシステムの設定
- **infinispan** サブシステムの設定
- ローカルの Hibernate キャッシュコンテナー



注記

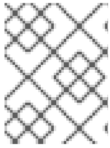
このレイヤーは **jpa-distributed** で配布されるレイヤーと互換性がありません。**jpa** レイヤーを含める場合は、**jpa-distributed** レイヤーを含めることはできません。

このレイヤーは、起動可能な JAR をビルドする場合のみサポートされます。このレイヤーは、S2I を使用する場合にはサポートされません。

jpa-distributed
このデコレーターレイヤーは、クラスターで稼働しているサーバーに永続性を追加します。レイヤーは Hibernate ライブラリーをプロビジョニングされたサーバーに追加し、以下のサポートを受けることができます。

- **jpa** サブシステムの設定
- **infinispan** サブシステムの設定
- ローカルの Hibernate キャッシュコンテナー
- インバリデーションおよびレプリケーション Hibernate キャッシュコンテナー

- **jgroups** サブシステムの設定



注記

このレイヤーは **jpa** レイヤーと互換性がありません。**jpa** レイヤーを含める場合は、**jpa-distributed** レイヤーを含めることはできません。

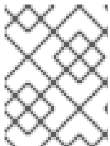
このレイヤーは、起動可能な JAR をビルドする場合のみサポートされます。このレイヤーは、S2I を使用する場合にはサポートされません。

Jakarta Server Faces
このデコレーターレイヤーは、プロビジョニングしたサーバーに **jsf** サブシステムを追加します。

このレイヤーは、起動可能な JAR をビルドする場合のみサポートされます。このレイヤーは、S2I を使用する場合にはサポートされません。

microprofile-platform
このデコレーターレイヤーは、以下の MicroProfile 機能を、プロビジョニングされたサーバーに追加します。

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

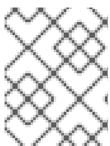


注記

このレイヤーには、**observability** レイヤーにも含まれる MicroProfile 機能が含まれます。このレイヤーを含める場合は、**observability** レイヤーを含める必要はありません。

可観測性
このデコレーターレイヤーは、プロビジョニングしたサーバーに以下の observability 機能を追加します。

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



注記

このレイヤーは、**cloud-server** レイヤーに組み込まれています。このレイヤーは **cloud-server** レイヤーに追加する必要はありません。

remote-activemq
このデコレーターレイヤーは、リモート ActiveMQ ブローカーと通信し、メッセージングサポートを統合する機能を追加します。

プールされた接続ファクトリー設定は、**guest** を **user** および **password** 属性の値として指定します。CLI スクリプトを使用して、起動時にこれらの値を変更できます。

このレイヤーは、起動可能な JAR をビルドする場合のみサポートされます。このレイヤーは、S2I を使用する場合にはサポートされません。

sso
このデコレーターレイヤーは、プロビジョニングしたサーバーに Red Hat Single Sign-On 統合を追加します。

このレイヤーは、S2I を使用してサーバーをプロビジョニングする場合にのみ使用してください。

web-console
このデコレーターレイヤーは、プロビジョニングしたサーバーに管理コンソールを追加します。

このレイヤーは、起動可能な JAR をビルドする場合のみサポートされます。このレイヤーは、S2I を使用する場合にはサポートされません。

web-clustering
このデコレーターレイヤーは、クラスタリング環境に適したデータセッション処理に、非ローカルの Infinispan ベースのコンテナー Web キャッシュを設定することにより、分散可能な Web アプリケーションをサポートします。

web-passivation
このデコレーターレイヤーは、単一ノード環境に適したデータセッション処理に、ローカルの Infinispan ベースのコンテナー Web キャッシュを設定することにより、分散可能な Web アプリケーションをサポートします。

このレイヤーは、起動可能な JAR をビルドする場合のみサポートされます。このレイヤーは、S2I を使用する場合にはサポートされません。

webservices
このレイヤーは Jakarta Web サービスデプロイメントをサポートするプロビジョニングされたサーバーに Web サービス機能を追加します。

このレイヤーは、起動可能な JAR をビルドする場合のみサポートされます。このレイヤーは、S2I を使用する場合にはサポートされません。

関連情報

- プールされた接続ファクトリーの属性

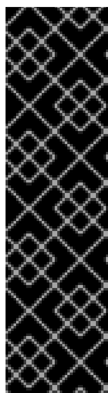
第7章 RED HAT CODEREADY STUDIO での JBOSS EAP MICROPROFILE アプリケーション開発の有効化

CodeReady Studio で開発するアプリケーションに MicroProfile 機能を組み込む場合は、CodeReady Studio で JBoss EAP の MicroProfile サポートを有効にする必要があります。

JBoss EAP 拡張パックは MicroProfile のサポートを提供します。

JBoss EAP 拡張パックは JBoss EAP 7.2 以前ではサポートされません。

JBoss EAP 拡張パックの各バージョンは、JBoss EAP の特定のパッチをサポートします。詳細は、JBoss EAP 拡張パックサポートおよびライフサイクルポリシーページを参照してください。



重要

JBoss EAP XP Quickstarts for Openshift はテクノロジープレビューとしてのみ提供されます。テクノロジープレビューの機能は、Red Hat の本番環境のサービスレベルアグリーメント (SLA) ではサポートされず、機能的に完全ではないことがあるため、Red Hat は本番環境での使用は推奨しません。これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、お客様は機能性をテストし、開発プロセス中にフィードバックをお寄せいただくことができます。

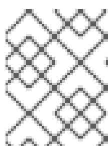
テクノロジープレビュー機能のサポート範囲については、Red Hat カスタマーポータルの テクノロジープレビュー機能のサポート範囲 を参照してください。

7.1. MICROPROFILE 機能を使用するための CODEREADY STUDIO の設定

JBoss EAP で MicroProfile サポートを有効にするには、JBoss EAP XP の新しいランタイムサーバーを登録し、新しい JBoss EAP 7.4 サーバーを作成します。

MicroProfile 機能をサポートすることを認識に役立つ適切な名前を付けます。

このサーバーは、以前にインストールされたランタイムを参照し、**standalone-microprofile.xml** 設定ファイルを使用する新たに作成された JBoss EAP XP ランタイムを使用します。



注記

Red Hat CodeReady Studio で **Target runtime** を **7.4** 以降に設定し、プロジェクトは Jakarta EE 8 仕様と互換性があります。

前提条件

- JBoss EAP XP 4.0.0 has been installed .

手順

1. **New Server** ダイアログボックスで新しいサーバーを設定します。
    a. **Select server type** リストで **Red Hat JBoss Enterprise Application Platform 7.4**を選択します。
    b. **Server's host name** フィールドに **localhost** を入力します。
    c. **Server name** フィールドに **JBoss EAP 7.4 XP** を入力します。

d. **次へ** をクリックします。

2. 新しいサーバーの設定

a. **Home directory** フィールドに、デフォルト設定を使用しない場合は、新しいディレクトリーを指定します (例: **home/myname/dev/microprofile/runtimes/jboss-eap-7.4**)。

b. **Ezecution Environment** が **JavaSE-1.8** に設定されていることを確認します。

c. 任意: **Server base directory** と **Configuration file** フィールドの値を変更します。

d. **Finish** をクリックします。

結果

これで、MicroProfile 機能を使用したアプリケーションの開発を開始することや、JBoss EAP の MicroProfile クイックスタートの使用できるようになりました。

7.2. CODEREADY STUDIO での MICROPROFILE クイックスタートの使用

MicroProfile クイックスタートを有効にすると、簡単な例はインストールされたサーバーで実行およびテストできるようになります。

これらの例は、以下の MicroProfile 機能を示しています。

- MicroProfile Config
- MicroProfile Fault Tolerance
- MicroProfile Health
- MicroProfile JWT
- MicroProfile Metrics
- MicroProfile OpenAPI
- MicroProfile OpenTracing
- MicroProfile REST クライアント

手順

1. Quickstart Parent Artifact から **pom.xml** ファイルをインポートします。
2. 使用しているクイックスタートで環境変数が必要な場合は、環境変数を設定します。
サーバーの **概要** ダイアログボックスで、起動設定に環境変数を定義します。

たとえば、**microprofile-opentracing** クイックスタートでは以下の環境変数を使用します。

- **JAEGER_REPORTER_LOG_spans** を **true** に設定
- **JAEGER_SAMPLER_PARAM** を **1** に設定
- **JAEGER_SAMPLER_TYPE** を **const** に設定

関連情報

Microprofile について

About JBoss Enterprise Application Platform expansion pack

Red Hat JBoss Enterprise Application Platform expansion pack サポートとライフサイクルポリシー

第8章 起動可能な JAR

JBoss EAP JAR Maven プラグインを使用して、マイクロサービスアプリケーションを起動可能な JAR としてビルドおよびパッケージ化できます。その後、JBoss EAP ベアメタルプラットフォームまたは JBoss EAP OpenShift プラットフォームでアプリケーションを実行できます。

8.1. 起動可能な JAR について

JBoss EAP JAR Maven プラグインを使用して、マイクロサービスアプリケーションを起動可能な JAR としてビルドおよびパッケージ化できます。

起動可能な JAR には、サーバー、パッケージ化されたアプリケーション、およびサーバー起動に必要なランタイムが含まれます。

JBoss JAR Maven プラグインは Galleon トリム機能を使用して、サーバーのサイズおよびメモリーフットプリントを削減します。そのため、必要な機能を提供する Galleon レイヤーのみを含め、要件に応じてサーバーを設定できます。

JBoss EAP JAR Maven プラグインは、サーバー設定をカスタマイズするための JBoss EAP CLI スクリプトファイルの実行をサポートします。CLI スクリプトには、サーバーを設定するための CLI コマンドのリストが含まれます。

起動可能な JAR は、以下の方法で標準の JBoss EAP サーバーと似ています。

- JBoss EAP の共通の管理 CLI コマンドをサポートします。
- JBoss EAP 管理コンソールを使用して管理できます。

起動可能な JAR でサーバーをパッケージ化する場合は、以下の制限が適用されます。

- サーバー再起動を必要とする CLI 管理操作はサポートされていません。
- サーバーは、サーバー管理に関連するサービスを開始するモードである admin-only モードで再起動できません。
- サーバーをシャットダウンすると、サーバーに設定した更新が失われます。

さらに、起動可能な hollow JAR をプロビジョニングできます。この JAR にはサーバーのみが含まれるため、サーバーを変更して異なるアプリケーションを実行することができます。

関連情報

機能のトリムに関する情報は、Capability Trimming を参照してください。

8.2. JBOSS EAP MAVEN プラグイン

JBoss EAP JAR Maven プラグインを使用してアプリケーションを起動可能な JAR としてビルドできます。

Maven リポジトリーから最新の Maven プラグインバージョンを取得できます。これは、/ga/org/wildfly/plugins/wildfly-jar-maven-plugin のインデックス にあります。

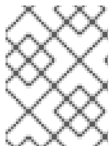
Maven プロジェクトで、**src** ディレクトリーにはアプリケーションのビルドに必要なすべてのソースファイルが含まれています。JBoss EAP JAR Maven プラグインが起動可能な JAR をビルドすると、生成された JAR は **target/<application>-bootable.jar** に置かれます。

JBoss EAP JAR Maven プラグインによって以下の機能も提供されます。

- CLI スクリプトコマンドをサーバーに適用します。
- サーバー設定ファイルをカスタマイズするには、**org.jboss.eap:wildfly-galleon-pack** Galleon 機能パックと、そのレイヤーの一部を使用します。
- キーストアファイルなど、パッケージ化可能な JAR に追加ファイルの追加をサポートします。
- 起動可能な hollow JAR を作成する機能が含まれています。つまり、アプリケーションを含まない起動可能な JAR です。

JBoss EAP JAR Maven プラグインを使用して起動可能な JAR を作成した後に、以下のコマンドを実行するとアプリケーションを起動できます。**target/myapp-bootable.jar** を起動可能な JAR へのパスに置き換えます。以下に例を示します。

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



注記

サポートされる起動可能な JAR 起動コマンドのリストを取得するには、起動コマンドの最後に **--help** を追加します。例: **java -jar target/myapp-bootable.jar --help**

関連情報

- サポートされる JBoss EAP Galleon レイヤーの詳細は、Available JBoss EAP layers を参照してください。
- プロジェクトの機能パックをビルドするためのサポート対象の Galleon プラグインの詳細は、WildFly Galleon Maven Plugin のドキュメント を参照してください。
- JBoss EAP Maven リポジトリーの設定方法の選択に関する詳細は、Maven リポジトリーの使用 を参照してください。
- Maven プロジェクトディレクトリーの詳細は、**Apache Maven** ドキュメントの Introduction to the Standard Directory Layout を参照してください。

8.3. 起動可能な JAR 引数

以下の表で引数を確認し、起動可能な JAR で使用するサ対応引数について確認します。

表8.1 対応の起動可能な JAR 実行引数

引数	説明
--help	指定のコマンドのヘルプメッセージを表示し、終了します。

引数	説明
--cli-script=<path>	起動可能な JAR の起動時に実行される JBoss CLI スクリプトへのパスを指定します。指定されたパスが相対パスである場合、パスは、起動可能な JAR の起動に使用される Java VM インスタンスの作業ディレクトリーに対して解決されます。
--deployment=<path>	起動可能な JAR に固有の引数。サーバーにデプロイするアプリケーションが含まれる WAR、JAR、EAR ファイル、またはデプロイメント形式のディレクトリーへのパスを指定します。
--display-galleon-config	生成された Galleon 設定ファイルの内容を出力します。
--install-dir=<path>	デフォルトでは、JVM 設定は、起動可能な JAR の起動後に **TEMP** ディレクトリーを作成するために使用されます。**--install-dir** 引数を使用するとサーバーをインストールするディレクトリーを指定できます。
-secmgr	セキュリティーマネージャーがインストールされた状態でサーバーを実行します。
-b<interface>=<value>	システムプロパティー **jboss.bind.address.<interface>** を指定の値に設定します。例: **bmanagement=IP_ADDRESS**
-b=<value>	パブリックインターフェイスのバインドアドレスを設定するために使用される **jboss.bind.address** システムプロパティーを設定します。値の指定がない場合はデフォルトで 127.0.0.1 が指定されます。

引数	説明
-D<name>[=<value>]	サーバーランタイムにサーバーによって設定されるシステムプロパティーを指定します。起動可能な JAR JVM はこれらのシステムプロパティーを設定しません。
--properties=<url>	指定の URL からシステムプロパティーをロードします。
-S<name>[=value]	セキュリティープロパティーを設定します。
-u=<value>	設定ファイルの socket-binding 要素のマルチキャストアドレスを設定するために使用される **jboss.default.multicast.address** システムプロパティーを設定します。値の指定がない場合はデフォルトで 230.0.0.4 が指定されます。
--version	アプリケーションサーバーのバージョンを表示し、終了します。

8.4. 起動可能な JAR サーバーの GALLEON レイヤーの指定

Galleon レイヤーを指定して、サーバーのカスタム設定をビルドできます。さらに、サーバーから除外する Galleon レイヤーを指定することもできます。

単一の機能パックを参照するには、**<feature-pack-location>** 要素を使用してその場所を指定します。以下の例は、Maven プラグイン設定ファイルの **<feature-pack-location>** 要素に **org.jboss.eap:wildfly-galleon-pack:4.0.0.GA-redhat-00002** を指定します。

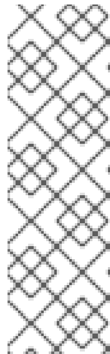
```
<configuration>
  <feature-pack-location>org.jboss.eap:wildfly-galleon-pack:4.0.0.GA-redhat-00002</feature-pack-
location>
</configuration>
```

複数の機能パックを参照する必要がある場合は、それらを **<feature-packs>** 要素に一覧表示します。以下の例は、Red Hat Single Sign-On 機能パックを **<feature-packs>** 要素に追加する方法を示しています。

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

```
            <location>org.keycloak:keycloak-adapter-galleon-pack:15.0.4.redhat-00001</location>
        </feature-pack>
    </feature-packs>
</configuration>
```

複数の機能パックから Galleon レイヤーを組み合わせて、起動可能な JAR サーバーを設定し、必要な機能を提供する対応の Galleon レイヤーのみを含めることができます。



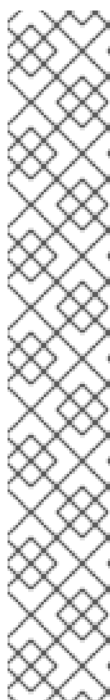
注記

ベアメタルプラットフォームで、設定ファイルで Galleon レイヤーを指定しない場合、プロビジョニングしたサーバーにはデフォルトの **standalone-microprofile.xml** 設定と同じ設定が含まれます。

OpenShift プラットフォームでは、プラグイン設定に **<cloud/>** 設定要素を追加した後に、設定ファイルで Galleon レイヤーを指定しない場合、プロビジョニングされたサーバーにはクラウド環境用に調整され、デフォルトの **standalone-microprofile-ha.xml** と似ている設定が含まれます。

前提条件

- Maven がインストールされている。
- **MAVEN_PLUGIN_VERSION.X.GA.Final-redhat-00001** などの最新の Maven プラグインバージョンを確認している。**MAVEN_PLUGIN_VERSION** はメジャーバージョンで、**X** はマイクロバージョンです。/ga/org/wildfly/plugins/wildfly-jar-maven-plugin のインデックス を参照してください。
- **4.0.X.GA-redhat-BUILD_NUMBER** などの最新の Galleon feature-pack バージョンを確認している。**X** は JBoss EAP XP のマイクロバージョンで、**BUILD_NUMBER** は Galleon 機能パックのビルド番号。JBoss EAP XP 4.0.0 製品のライフサイクル中に、**X** と **BUILD_NUMBER** の両方を進化できる。Index of /ga/org/jboss/eap/wildfly-galleon-pack のインデックス を参照してください。



注記

この手順の例では、以下のプロパティーを指定します。

- Maven プラグインバージョンの場合は、**${bootable.jar.maven.plugin.version}** です。
- Galleon 機能パックバージョンの場合は、**${jboss.xp.galleon.feature.pack.version}** です。

これらのプロパティーをプロジェクトで設定する必要があります。以下に例を示します。

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

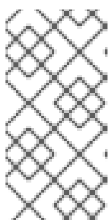
手順

1. アプリケーションの実行に必要な機能を提供する、サポートされる JBoss EAP Galleon レイヤーを特定します。

2. Maven プロジェクトの **pom.xml** ファイルの **<plugin>** 要素で JBoss EAP feature-pack の場所を参照します。以下の例のように、最新バージョンの Maven プラグインと **org.jboss.eap:wildfly-galleon-pack** Galleon 機能パックの最新バージョンを指定する必要があります。以下の例は、**jaxrs-server** ベースレイヤーおよび **jpa-distribute** レイヤーを含む単一の feature-pack も含まれています。**jaxrs-server** ベースレイヤーは、サーバーの追加サポートを提供します。

```
<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>
```

この例では、プロジェクトからの **jpa** レイヤーの除外も示しています。



注記

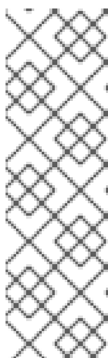
jpa-distributed レイヤーをプロジェクトに含める場合は、**jaxrs-server** レイヤーから **jpa** レイヤーを除外する必要があります。**jpa** レイヤーはローカルの infinispan hibernate キャッシュを設定し、**jpa-distributed** レイヤーはリモート infinispan hibernate キャッシュを設定します。

関連情報

- 利用可能なベースレイヤーの詳細は、Base layers を参照してください。
- プロジェクトの機能パックをビルドするためのサポート対象の Galleon プラグインの詳細は、WildFly Galleon Maven Plugin のドキュメント を参照してください。
- JBoss EAP Maven リポジトリーの設定方法の選択に関する詳細は、Maven and the JBoss EAP MicroProfile Maven repository を参照してください。
- Maven 依存関係の管理に関する詳細は、**Apache Maven Project** ドキュメントの Dependency Management を参照してください。

8.5. JBOSS EAP ベアメタルプラットフォームでの起動可能な JAR の使用

JBoss EAP ベアメタルプラットフォームで、アプリケーションを起動可能な JAR としてパッケージ化できます。



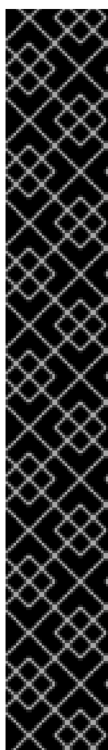
注記

- JBoss EAP ベアメタルプラットフォームで起動可能な JAR を構築するときにカスタム Galleon 機能パックとレイヤーを使用するには、JBoss EAP の カスタム Galleon レイヤーの構築と使用 を参照してください。
- **oc new-build** コマンドを使用してアプリケーションイメージをビルドする場合は、jboss-eap **74- openjdk11-openshift:latest ではなく、この S2I ビルダーイメージ jboss-eap-xp4 -openjdk11-openshift:latest** を使用するようにしてください。

起動可能な JAR には、サーバー、パッケージ化されたアプリケーション、およびサーバー起動に必要なランタイムが含まれます。

この手順では、JBoss EAP JAR Maven プラグインを使用して MicroProfile Config マイクロサービスを起動可能な JAR としてパッケージ化する方法を説明します。MicroProfile Config development を参照してください。

起動可能な JAR のパッケージング時に CLI スクリプトを使用してサーバーを設定できます。



重要

起動可能な JAR 内にパッケージ化する必要がある Web アプリケーションのビルドでは、**pom.xml** ファイルの **<packaging>** 要素に **war** を指定する必要があります。以下に例を示します。

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

この値は、ビルドアプリケーションを、デフォルトの JAR ファイルとしてではなく、WAR ファイルとしてパッケージ化するのに必要です。

起動可能な hollow JAR をビルドするためだけに Maven プロジェクトで使用される Maven プロジェクトで、packaging の値を **pom** に設定します。以下に例を示します。

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

Maven プロジェクトの hollow JAR をビルドする場合は、**pom** パッケージングの使用に限定されません。**war** などのパッケージ化の種類について **<hollow-jar>** 要素に **true** を指定すると作成できます。Creating a hollow bootable JAR on a JBoss EAP bare-metal platform を参照してください。

前提条件

- **MAVEN_PLUGIN_VERSION.X.GA.Final-redhat-00001** などの最新の Maven プラグインバージョンを確認している。**MAVEN_PLUGIN_VERSION** はメジャーバージョンで、**X** はマイクロバージョンです。/ga/org/wildfly/plugins/wildfly-jar-maven-plugin のインデックス を参照してください。
- **4.0.X.GA-redhat-BUILD_NUMBER** などの最新の Galleon feature-pack バージョンを確認している。**X** は JBoss EAP XP のマイクロバージョンで、 **BUILD_NUMBER** は Galleon 機能パックのビルド番号。JBoss EAP XP 4.0.0 製品のライフサイクル中に、**X** と **BUILD_NUMBER** の両

方を進化できる。Index of /ga/org/jboss/eap/wildfly-galleon-pack のインデックス を参照してください。

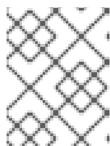
- Maven プロジェクトを作成し、親依存関係を設定して、MicroProfile アプリケーションを作成するための依存関係を追加している。MicroProfile Config development を参照してください。

> **注記**
>
> この手順の例では、以下のプロパティーを指定します。
>
> - Maven プラグインバージョンの場合は、**${bootable.jar.maven.plugin.version}** です。
> - Galleon 機能パックバージョンの場合は、**${jboss.xp.galleon.feature.pack.version}** です。
>
> これらのプロパティーをプロジェクトで設定する必要があります。以下に例を示します。
>
> ```
> <properties>
>     <bootable.jar.maven.plugin.version>6.1.2.Final-redhat-00001</bootable.jar.maven.plugin.version>
>     <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-00002</jboss.xp.galleon.feature.pack.version>
> </properties>
> ```

手順

1. 以下の内容を **pom.xml** ファイルの **<build>** 要素に追加します。最新バージョンの Maven プラグインと、**org.jboss.eap:wildfly-galleon-pack** Galleon 機能パックの最新バージョンを指定する必要があります。以下に例を示します。

```
<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>
```



注記

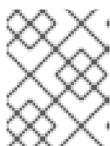
pom.xml ファイルで Galleon レイヤーを指定しない場合、起動可能な JAR サーバーには **standalone-microprofile.xml** 設定と同じ設定が含まれます。

2. アプリケーションを起動可能な JAR としてパッケージ化します。

```
$ mvn package
```

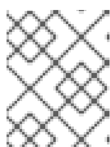
3. アプリケーションを起動します。

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



注記

この例では、環境変数に **NAME** を使用しますが、デフォルト値の **jim** を使用することができます。



注記

サポートされる起動可能な JAR 引数のリストを表示するには、**--help** を **java -jar target/microprofile-config-bootable.jar** コマンドの最後に追加します。

4. Web ブラウザーで以下の URL を指定して MicroProfile Config アプリケーションにアクセスします。

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

5. **検証:** ターミナルで以下のコマンドを実行し、アプリケーションが適切に動作します。

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

以下が想定される出力です。

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

関連情報

- 使用できる MicroProfile Config 機能の詳細は、MicroProfile Config を参照してください。
- **ConfigSources** の詳細は、MicroProfile Config reference を参照してください。

8.6. JBOSS EAP ベアメタルプラットフォームでの起動可能な JAR の作成

JBoss EAP ベアメタルプラットフォームで、アプリケーションを起動可能な hollow JAR としてパッケージ化できます。

起動可能な hollow JAR には JBoss EAP サーバーのみが含まれます。起動可能な hollow JAR は JBoss EAP JAR Maven プラグインによってパッケージ化されます。アプリケーションはサーバーランタイム時に提供されます。起動可能な JAR は、異なるアプリケーションのサーバー設定を再利用する必要がある場合に便利です。

前提条件

- Maven プロジェクトを作成し、親依存関係を設定して、アプリケーションを作成するための依存関係を追加している。MicroProfile Config development を参照してください。
- Using a bootable JAR on a JBoss EAP bare-metal platform で説明されている **pom.xml** ファイル設定手順を完了してください。
- **MAVEN_PLUGIN_VERSION.X.GA.Final-redhat-00001** などの最新の Maven プラグインバージョンを確認している。**MAVEN_PLUGIN_VERSION** はメジャーバージョンで、**X** はマイクロバージョンです。/ga/org/wildfly/plugins/wildfly-jar-maven-plugin のインデックス を参照してください。
- **4.0.X.GA-redhat-BUILD_NUMBER** などの最新の Galleon feature-pack バージョンを確認している。**X** は JBoss EAP XP のマイクロバージョンで、 **BUILD_NUMBER** は Galleon 機能パックのビルド番号。JBoss EAP XP 4.0.0 製品のライフサイクル中に、**X** と **BUILD_NUMBER** の両方を進化できる。Index of /ga/org/jboss/eap/wildfly-galleon-pack のインデックス を参照してください。

注記

この手順の例では、Galleon 機能パックバージョンに **${jboss.xp.galleon.feature.pack.version}** を指定しますが、プロジェクトでプロパティーを設定する必要があります。以下に例を示します。

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

手順

1. 起動可能な hollow JAR をビルドするには、プロジェクトの **pom.xml** ファイルで **<hollow-jar>** プラグイン設定要素を true に設定する必要があります。以下に例を示します。

```
<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>
```

注記

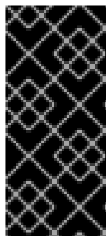
<hollow-jar> 要素で **true** を指定すると、JBoss EAP JAR Maven プラグインにアプリケーションが含まれません。

1. 起動可能な hollow JAR をビルドします。

```
$ mvn clean package
```

2. 起動可能な hollow JAR を実行します。

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



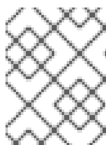
重要

サーバーにデプロイする WAR ファイルへのパスを指定するには、以下の引数を使用します。**<PATH_NAME>** は、デプロイメントへのパスになります。

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

3. アプリケーションにアクセスします。

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



注記

root ディレクトリーに Web アプリケーションを登録するには、アプリケーションに **ROOT.war** という名前を付けます。

関連情報

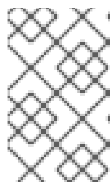
- 使用できる MicroProfile 機能は、MicroProfile Config を参照してください。
- JBoss EAP XP 4.0.0 でサポートされる JBoss EAP JAR Maven プラグインの詳細は、JBoss EAP Maven plug-in を参照してください。

8.7. ビルド時に実行される CLI スクリプト

CLI スクリプトを作成して、起動可能な JAR のパッケージング中にサーバーを設定できます。

CLI スクリプトは、追加のサーバー設定を適用するために使用できる一連の CLI コマンドが含まれるテキストファイルです。たとえば、スクリプトを作成して新しいロガーを **logging** サブシステムに追加できます。

CLI スクリプトでより複雑な操作を指定することもできます。たとえば、セキュリティー管理操作を単一のコマンドにグループ化して、管理 HTTP エンドポイントの HTTP 認証を有効にできます。



注記

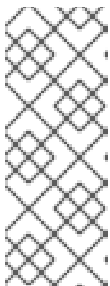
アプリケーションを起動可能な JAR としてパッケージ化する前に、プラグイン設定の **<cli-session>** 要素に CLI スクリプトを定義する必要があります。これにより、起動可能な JAR をパッケージ化した後にサーバー設定が維持されるようになります。

事前定義された Galleon レイヤーを組み合わせて、アプリケーションをデプロイするサーバーを設定できますが、制限はあります。たとえば、起動可能な JAR のパッケージ化時に Galleon レイヤーを使用して HTTPS **undertow** リスナーを有効にすることはできません。代わりに、CLI スクリプトを使用する必要があります。

CLI スクリプトは、**pom.xml** ファイルの **<cli-session>** 要素に定義する必要があります。以下の表は、CLI セッション属性のタイプを示しています。

表8.2 CLI スクリプトの属性

引数	説明
script-files	スクリプトファイルへのパスのリスト。
properties-file	プロパティーファイルへのパスを指定する任意の属性。このファイルは、**${my.prop}** 構文を使用してスクリプトが参照できる Java プロパティーをリスト表示します。以下の例では、**public inet-address** を **all.addresses** の値に設定します。**/interface=public:write-attribute(name=inet-address,value=${all.addresses})**
resolve-expressions	ブール値が含まれる任意の属性です。操作リクエストをサーバーに送信する前にシステムプロパティーまたは式が解決されているかどうかを示します。デフォルト値は **true** です。



注記

- CLI スクリプトは、**pom.xml** ファイルの **<cli-session>** 要素で定義された順序で起動します。
- JBoss EAP JAR Maven プラグインは、各 CLI セッションに対して埋め込みサーバーを起動します。そのため、CLI スクリプトは埋め込みサーバーを起動したり、停止したりする必要はありません。

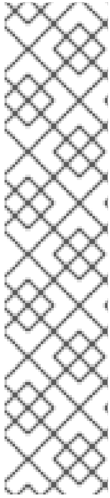
8.8. 実行時に CLI スクリプトを実行する

実行時にサーバー設定に変更を適用できます。これにより、実行コンテキストに関してサーバーを柔軟に調整できます。ただし、サーバーに変更を適用するための推奨される方法は、ビルド時です。

手順

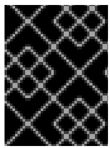
- 起動可能な JAR と **--cli-script** 引数を起動します。
  以下に例を示します。

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



注記

- CLI スクリプトはテキストファイル (UTF-8) である必要があります。ファイル拡張子が存在する場合は意味がありませんが、**.cli** 拡張子を使用することを推奨します。
- サーバーの再起動が必要な操作は、起動可能な JAR インスタンスを終了します。
- **connect**、**reload**、**shutdown** などの CLI コマンド、および組み込みサーバーと **patch** に関連するコマンドは機能しません。
- admin モードでは実行できない **jdbc-driver-info** などの CLI コマンドはサポートされていません。

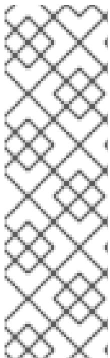


重要

CLI スクリプトを実行せずにサーバーを再起動すると、新しいサーバーインスタンスには以前のサーバーインスタンスからの変更が含まれなくなります。

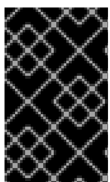
8.9. JBOSS EAP OPENSHIFT プラットフォームでの起動可能な JAR の使用

アプリケーションを起動可能な JAR としてパッケージ化した後、JBoss EAP OpenShift プラットフォームでアプリケーションを実行できます。



注記

- JBoss EAP OpenShift プラットフォームで起動可能な JAR を構築するときにカスタム Galleon 機能パックとレイヤーを使用するには、JBoss EAP カスタム Galleon レイヤーの構築と使用 を参照してください。
- **oc new-build** コマンドを使用してアプリケーションイメージをビルドする場合は、jboss-eap **74- openjdk11-openshift:latest ではなく、この S2I ビルダーイメージ jboss-eap-xp4 -openjdk11-openshift:latest** を使用するようにしてください。



重要

OpenShift では、起動可能な JAR で EAP Operator の自動化トランザクションリカバリー機能を使用することはできません。この技術制限の修正は、今後の JBoss EAP XP 4.0.0 パッチリリースに対して予定されています。

前提条件

- MicroProfile Config development 用の Maven プロジェクトを作成している。
- **MAVEN_PLUGIN_VERSION.X.GA.Final-redhat-00001** などの最新の Maven プラグインバージョンを確認している。**MAVEN_PLUGIN_VERSION** はメジャーバージョンで、**X** はマイクロバージョンです。/ga/org/wildfly/plugins/wildfly-jar-maven-plugin のインデックス を参照してください。
- **4.0.X.GA-redhat-BUILD_NUMBER** などの最新の Galleon feature-pack バージョンを確認している。**X** は JBoss EAP XP 4 のマイクロバージョンで、 **BUILD_NUMBER** は Galleon 機能パッ

クのビルド番号である。JBoss EAP XP 4.0.0 製品のライフサイクル中に、**X** と **BUILD_NUMBER** の両方を進化できる。Index of /ga/org/jboss/eap/wildfly-galleon-pack のインデックス を参照してください。

注記

この手順の例では、以下のプロパティーを指定します。

- Maven プラグインバージョンの場合は、**${bootable.jar.maven.plugin.version}** です。
- Galleon 機能パックバージョンの場合は、**${jboss.xp.galleon.feature.pack.version}** です。

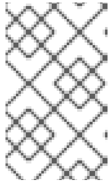
これらのプロパティーをプロジェクトで設定する必要があります。以下に例を示します。

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

手順

1. 以下の内容を **pom.xml** ファイルの **<build>** 要素に追加します。最新バージョンの Maven プラグインと、**org.jboss.eap:wildfly-galleon-pack** Galleon 機能パックの最新バージョンを指定する必要があります。以下に例を示します。

```
<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>
```



注記

<cloud/> 要素をプラグイン設定の **<configuration>** 要素に含める必要があります。そのため、JBoss EAP Maven JAR プラグインは OpenShift プラットフォームを選択できます。

2. アプリケーションをパッケージ化します。

```
$ mvn package
```

3. **oc login** コマンドを使用して、OpenShift インスタンスにログインします。

4. OpenShift で新しいプロジェクトを作成します。以下に例を示します。

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

5. 以下の **oc** コマンドを入力してアプリケーションイメージを作成します。

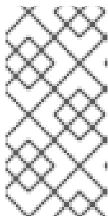
```
$ mkdir target/openshift && cp target/microprofile-config-bootable.jar target/openshift (1)

$ 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) ターゲットディレクトリーに openshift サブディレクトリーを作成します。パッケージ化されたアプリケーションが、作成されたサブディレクトリーにコピーされます。

(2) 最新の OpenJDK 11 イメージストリームタグおよびイメージ情報を OpenShift プロジェクトにインポートします。

(3) microprofile-config-app ディレクトリーおよび OpenJDK 11 イメージストリームに基づいてビルド設定を作成します。

(4) **target/openshift** サブディレクトリーをバイナリー入力として使用し、アプリケーションをビルドします。



注記

OpenShift は CLI スクリプトコマンドのセットを起動可能な JAR 設定ファイルに適用し、クラウド環境に合わせて調整します。このスクリプトにアクセスするには、Maven プロジェクト **/target directory** の **bootable-jar-build-artifacts/generated-cli-script.txt** ファイルを開きます。

6. **検証:**
利用可能な OpenShift Pod のリストを表示し、以下のコマンドを実行して Pod のビルドステータスを確認します。

```
$ oc get pods
```

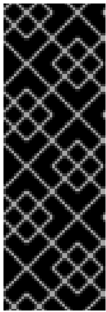
ビルドされたアプリケーションイメージを確認します。

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

出力には、名前、イメージリポジトリー、タグなどのビルドされたアプリケーションイメージの詳細が表示されます。この手順の例では、イメージストリーム名とタグの出力には **microprofile-config-app:latest** が表示されます。

7. アプリケーションのデプロイ:

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

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



重要

起動可能な JAR にシステムプロパティーを指定するには、**JAVA_OPTS_APPEND** 環境変数を使用する必要があります。以下の例は、**JAVA_OPTS_APPEND** 環境変数の使用方法を示しています。

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

新しいアプリケーションが作成され、起動します。アプリケーション設定は新しいサービスとして公開されます。

8. **検証**: ターミナルで以下のコマンドを実行し、アプリケーションが適切に動作するかどうかをテストします。

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

想定される出力:

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

関連情報

- MicroProfile の詳細は、MicroProfile Config を参照してください。
- **ConfigSources** の詳細は、Default MicroProfile Config attributes を参照してください。

8.10. OPENSHIFT の起動可能な JAR の設定

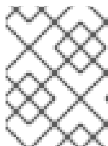
起動可能な JAR を使用する前に、JVM を設定してスタンドアロンサーバーが JBoss EAP for OpenShift で正しく動作することを確認できます。

JAVA_OPTS_APPEND 環境変数を使用して JVM を設定します。**JAVA_ARGS** コマンドを使用して、起動可能な JAR に引数を提供します。

環境変数を使用してプロパティーの値を設定できます。たとえば、**JAVA_OPTS_APPEND** 環境変数を使用して、**-Dwildfly.statistics-enabled** プロパティーを **true** に設定できます。

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

サーバーの統計が有効になっているようになりました。



注記

起動可能な JAR に引数を提供する必要がある場合は、**JAVA_ARGS** 環境変数を使用します。

JBoss EAP for OpenShift は JDK 11 イメージを提供します。起動可能な JAR に関連付けられたアプリケーションを実行するには、まず最新の OpenJDK 11 イメージストリームタグおよびイメージ情報を OpenShift プロジェクトにインポートする必要があります。環境変数を使用して、インポートされたイメージで JVM を設定できます。

JBoss EAP for OpenShift S2I イメージに使用される JVM の設定に同じ設定オプションを適用できますが、以下の違いがあります。

- オプション: **-Xlog** 機能は利用できませんが、**-Xlog:gc** を有効にすることでガベッジコレクションのロギングを設定できます。例: **JAVA_OPTS_APPEND="-Xlog:gc*:file=/tmp/gc.log:time"**
- 初期メタスペースのサイズを増やすには、**GC_METASPACE_SIZE** 環境変数を設定します。最適なメタデータのキャパシティパフォーマンスを得るためには、値を **96** に設定します。
- ランダムなファイルの生成を改善するには、**JAVA_OPTS_APPEND** 環境変数を使用して、**java.security.egd** プロパティーを **-Djava.security.egd=file:/dev/urandom** に設定します。

この設定により、インポートされた OpenJDK 11 イメージで実行される場合に JVM のメモリー設定およびガベージコレクション機能が向上します。

8.11. OPENSHIFT でのアプリケーションでの CONFIGMAP の使用

OpenShift では、デプロイメントコントローラー (dc) を使用して、アプリケーションの実行に使用される Pod に configmap をマウントできます。

ConfigMap は、機密ではないデータをキーと値のペアに保存するために使用される OpenShift リソースです。

microprofile-platform Galleon レイヤーを指定して **microprofile-config-smallrye** およびすべての拡張機能をサーバー設定に追加してから、CLI スクリプトを使用して新しい **ConfigSource** をサーバー設定に追加できます。CLI スクリプトは、Maven プロジェクトのルートディレクトリーにある **/scripts** ディレクトリーなど、アクセス可能なディレクトリーに保存できます。

MicroProfile Config 機能は、SmallRye Config を使用して JBoss EAP に実装され、**microprofile-config-smallrye** サブシステムによって提供されます。このサブシステムは **microprofile-platform** Galleon レイヤーに含まれています。

前提条件

- Maven がインストールされている。
- JBoss EAP Maven リポジトリーを設定している。
- アプリケーションを起動可能な JAR としてパッケージ化し、JBoss EAP OpenShift プラットフォームでアプリケーションを実行できます。OpenShift プラットフォーム上でアプリケー

ションを起動可能な JAR としてビルドする方法は、Using a bootable JAR on a JBoss EAP OpenShift platform を参照してください。

手順

1. プロジェクトのルートディレクトリーに **scripts** という名前のディレクトリーを作成します。以下に例を示します。

```
$ mkdir scripts
```

2. **cli.properties** ファイルを作成し、そのファイルを **/scripts** ディレクトリーに保存します。このファイルに **config.path** および **config.ordinal** システムプロパティーを定義します。以下に例を示します。

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

3. **mp-config.cli** などの CLI スクリプトを作成し、これを **/scripts** ディレクトリーなどの起動可能な JAR のアクセス可能なディレクトリーに保存します。以下の例は、**mp-config.cli** スクリプトの内容を示しています。

```
# config map

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

mp-config.cli CLI スクリプトは、新しい **ConfigSource** を作成し、Pipelineal および path の値をプロパティーファイルから取得します。

4. スクリプトを **/scripts** ディレクトリーに保存します。このディレクトリーは、プロジェクトのルートディレクトリーにあります。

5. 既存のプラグイン **<configuration>** 要素に以下の設定抽出を追加します。

```
<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. アプリケーションをパッケージ化します。

```
$ mvn package
```

7. **oc login** コマンドを使用して、OpenShift インスタンスにログインします。

8. **オプション: target/openshift** サブディレクトリーを作成していない場合は、以下のコマンドを実行してサブディレクトリーを作成する必要があります。

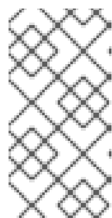
```
$ mkdir target/openshift
```

9. パッケージ化したアプリケーションを、作成したサブディレクトリーにコピーします。

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

10. **target/openshift** サブディレクトリーをバイナリー入力として使用し、アプリケーションをビルドします。

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



注記

OpenShift は CLI スクリプトコマンドのセットを起動可能な JAR 設定ファイルに適用し、クラウド環境に合わせて調整します。このスクリプトにアクセスするには、Maven プロジェクト **/target directory** の **bootable-jar-build-artifacts/generated-cli-script.txt** ファイルを開きます。

11. **ConfigMap** を作成します。以下に例を示します。

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

12. dc を使用して、**ConfigMap** をアプリケーションにマウントします。以下に例を示します。

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

oc set volume コマンドを実行すると、アプリケーションは新しい設定で再デプロイされます。

13. 出力をテストします。

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

以下が想定される出力です。

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

関連情報

- MicroProfile Config **ConfigSources** 属性の詳細は、Default MicroProfile Config attributes を参照してください。
- 起動可能な JAR 引数の情報は、Supported bootable JAR arguments を参照してください。

8.12. 起動可能な JAR MAVEN プロジェクトの作成

以下の手順に従って、サンプル Maven プロジェクトを作成します。以下の手順を実行する前に、Maven プロジェクトを作成する必要があります。

- 起動可能な JAR の JSON ロギングの有効化
- 複数の起動可能な JAR インスタンスの Web セッションデータストレージの有効化
- CLI スクリプトを使用した起動可能な JAR の HTTP 認証の有効化
- Red Hat Single Sign-On での JBoss EAP の起動可能な JAR アプリケーションのセキュア化

プロジェクトの **pom.xml** ファイルでは、起動可能な JAR のビルドに必要なプロジェクトアーティファクトを取得するように Maven を設定できます。

手順

1. Maven プロジェクトを設定します。

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

   GROUP_ID はプロジェクトの **groupId** で、**ARTIFACT_ID** はプロジェクトの **artifactId** です。

2. **pom.xml** ファイルで、リモートリポジトリーから JBoss EAP BOM ファイルを取得するように Maven を設定します。

```
<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. **jboss-eap-jakartaee8** BOM の Jakarta EE アーティファクトのバージョンを自動的に管理するように Maven を設定するには、プロジェクトの **pom.xml** ファイルの **<dependencyManagement>** セクションに BOM を追加します。以下に例を示します。

```
<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. 以下の例のように、BOM によって管理されるサーブレット API アーティファクトをプロジェクトの **pom.xml** ファイルの **<dependency>** セクションに追加します。

```
<dependency>
    <groupId>org.jboss.spec.javax.servlet</groupId>
    <artifactId>jboss-servlet-api_4.0_spec</artifactId>
    <scope>provided</scope>
</dependency>
```

関連情報

- JBoss EAP Maven プラグインの詳細は、JBoss EAP Maven plug-in を参照してください。
- Galleon レイヤーの詳細は、Specifying Galleon layers for your bootable JAR server を参照してください。
- プロジェクトで Red Hat Single Sign-On Galleon 機能パックを含める方法は、Securing your JBoss EAP bootable JAR application with Red Hat Single Sign-On を参照してください。

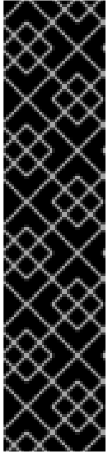
8.13. 起動可能な JAR の JSON ロギングの有効化

CLI スクリプトを使用してサーバーロギング設定を設定すると、起動可能な JAR の JSON ロギングを有効にできます。JSON ロギングを有効にすると、JSON フォーマッターを使用してログメッセージを JSON 形式で表示できます。

この手順の例では、ベアメタルプラットフォームおよび OpenShift プラットフォームで、起動可能な JAR の JSON ロギングを有効にする方法を説明します。

前提条件

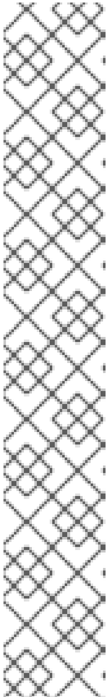
- **MAVEN_PLUGIN_VERSION.X.GA.Final-redhat-00001** などの最新の Maven プラグインバージョンを確認している。**MAVEN_PLUGIN_VERSION** はメジャーバージョンで、**X** はマイクロバージョンです。/ga/org/wildfly/plugins/wildfly-jar-maven-plugin のインデックス を参照してください。
- **4.0.X.GA-redhat-BUILD_NUMBER** などの最新の Galleon 機能パックバージョンを確認している。**X** は JBoss EAP XP のマイナーバージョンで、**BUILD_NUMBER** は Galleon 機能パックのビルド番号。JBoss EAP XP 4.0.0 製品のライフサイクル中に、**X** と **BUILD_NUMBER** の両方を進化できる。Index of /ga/org/jboss/eap/wildfly-galleon-pack のインデックス を参照してください。
- Maven プロジェクトを作成し、親依存関係を設定して、アプリケーションを作成するための依存関係を追加している。Creating a bootable JAR Maven project を参照してください。



重要

Maven プロジェクトの Maven archetype で、プロジェクト固有の groupID および artifactID を指定する必要があります。以下に例を示します。

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



注記

この手順の例では、以下のプロパティーを指定します。

- Maven プラグインバージョンの場合は、**${bootable.jar.maven.plugin.version}** です。
- Galleon 機能パックバージョンの場合は、**${jboss.xp.galleon.feature.pack.version}** です。

これらのプロパティーをプロジェクトで設定する必要があります。以下に例を示します。

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

手順

1. BOM によって管理される JBoss Logging および Jakarta RESTful Web Services 依存関係を、プロジェクトの **pom.xml** ファイルの **<dependencies>** セクションに追加します。以下に例を示します。

```
<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. 以下の内容を **pom.xml** ファイルの **<build>** 要素に追加します。最新バージョンの Maven プラグインと、**org.jboss.eap:wildfly-galleon-pack** Galleon 機能パックの最新バージョンを指定する必要があります。以下に例を示します。

```
<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. java ファイルを保存するディレクトリーを作成します。

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

APPLICATION_ROOT は、アプリケーションの **pom.xml** 設定ファイルが含まれるディレクトリーです。

4. 以下の内容で **Java ファイル RestApplication.java** を作成し、ファイルを **APPLICATION_ROOT/src/main/java/com/example/logging/** ディレクトリーに保存します。

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

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

5. 以下の内容で Java ファイル **HelloWorldEndpoint.java** を作成し、ファイルを **APPLICATION_ROOT/src/main/java/com/example/logging/** ディレクトリーに保存します。

```
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. **configure-oidc.cli** などの CLI スクリプトを作成し、**APPLICATION_ROOT/scripts** ディレクトリーなどの起動可能な JAR のアクセス可能なディレクトリーに保存します。**APPLICATION_ROOT** は Maven プロジェクトのルートディレクトリーです。スクリプトには以下のコマンドが含まれている必要があります。

```
/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. プラグイン **<configuration>** 要素に以下の設定抽出を追加します。

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

この例は、アプリケーションの JSON **ロギングを有効にするためにサーバーロギング設定ファイルを変更する logging.cli** CLI スクリプトを示しています。

8. アプリケーションを起動可能な JAR としてパッケージ化します。

```
$ mvn package
```

9. **オプション**: JBoss EAP ベアメタルプラットフォームでアプリケーションを実行するには、Using a bootable JAR on a JBoss EAP bare-metal platform にある手順に従いますが、以下の違いがあります。

   a. アプリケーションを起動します。

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

   b. 検証: ブラウザーで http://127.0.0.1:8080/hello に URL を指定すると、アプリケーションにアクセスできます。

予期される出力: アプリケーションコンソールで **com.example.logging.HelloWorldEndpoint** デバッグトレースを含む JSON 形式のログを表示できます。

10. **オプション**: JBoss EAP OpenShift プラットフォームでアプリケーションを実行するには、以下の手順を実行します。

    a. **<cloud/>** 要素をプラグイン設定に追加します。以下に例を示します。

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

    b. アプリケーションをリビルドします。

    ```
    $ mvn clean package
    ```

    c. **oc login** コマンドを使用して、OpenShift インスタンスにログインします。

    d. OpenShift で新しいプロジェクトを作成します。以下に例を示します。

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

    e. 以下の **oc** コマンドを入力してアプリケーションイメージを作成します。

    ```
    $ 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. **target/openshift** サブディレクトリーを作成します。パッケージ化されたアプリケーションは **openshift** サブディレクトリーにコピーされます。
    2. 最新の OpenJDK 11 イメージストリームタグおよびイメージ情報を OpenShift プロジェクトにインポートします。
    3. ロギングディレクトリーおよび OpenJDK 11 イメージストリームに基づいてビルド設定を作成します。
    4. **target/openshift** サブディレクトリーをバイナリー入力として使用し、アプリケーションをビルドします。

    f. アプリケーションのデプロイ:

```
$ oc new-app logging

$ oc expose svc/logging
```

g. ルートの URL を取得します。

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

h. 直前のコマンドから返された URL を使用して、Web ブラウザーでアプリケーションにアクセスします。以下に例を示します。

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

i. **Verfication:** 以下のコマンドを実行して、利用可能な OpenShift Pod のリストを表示し、Pod のビルドステータスを確認します。

```
$ oc get pods
```

アプリケーションの実行中の Pod ログにアクセスします。**APP_POD_NAME** は、実行中の Pod ロギングアプリケーションの名前です。

```
$ oc logs APP_POD_NAME
```

想定される結果: Pod ログは JSON 形式であり、**com.example.logging.HelloWorldEndpoint** デバッグトレースが含まれます。

関連情報

- JBoss EAP のロギング機能に関する詳細は、**設定ガイド**の JBoss EAP を用いたロギング を参照してください。
- OpenShift で起動可能な JAR を使用する方法は、Using a bootable JAR on a JBoss EAP OpenShift platform を参照してください。
- プロジェクトに JBoss EAP JAR Maven を指定する方法は、Specifying Galleon layers for your bootable JAR server を参照してください。
- CLI スクリプトの作成に関する詳細は、CLI Scripts を参照してください。

8.14. 複数の起動可能な JAR インスタンスの WEB セッションデータストレージの有効化

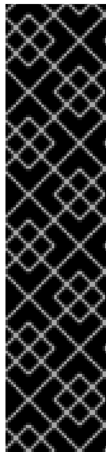
web クラスター化アプリケーションを起動可能な JAR としてビルドおよびパッケージ化できます。

前提条件

- **MAVEN_PLUGIN_VERSION.X.GA.Final-redhat-00001** などの最新の Maven プラグインバージョンを確認している。**MAVEN_PLUGIN_VERSION** はメジャーバージョンで、**X** はマイクロバージョンです。/ga/org/wildfly/plugins/wildfly-jar-maven-plugin のインデックス を参照してください。
- **4.0.X.GA-redhat-BUILD_NUMBER** などの最新の Galleon feature-pack バージョンを確認している。**X** は JBoss EAP XP のマイクロバージョンで、 **BUILD_NUMBER** は Galleon 機能パック

のビルド番号。JBoss EAP XP 4.0.0 製品のライフサイクル中に、**X** と **BUILD_NUMBER** の両方を進化できる。Index of /ga/org/jboss/eap/wildfly-galleon-pack のインデックス を参照してください。

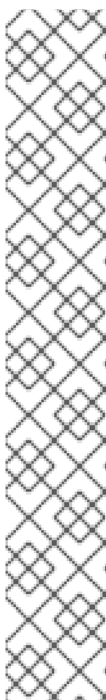
- Maven プロジェクトを作成し、親依存関係を設定して、web-clustering アプリケーションを作成するための依存関係を追加している。Creating a bootable JAR Maven project を参照してください。



重要

Maven プロジェクトを設定する場合は、Maven archetype 設定で値を指定する必要があります。以下に例を示します。

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



注記

この手順の例では、以下のプロパティーを指定します。

- Maven プラグインバージョンの場合は、**${bootable.jar.maven.plugin.version}** です。
- Galleon 機能パックバージョンの場合は、**${jboss.xp.galleon.feature.pack.version}** です。

これらのプロパティーをプロジェクトで設定する必要があります。以下に例を示します。

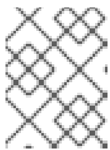
```
<properties>
    <bootable.jar.maven.plugin.version>6.1.2.Final-redhat-00001</bootable.jar.maven.plugin.version>
    <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-00002</jboss.xp.galleon.feature.pack.version>
</properties>
```

手順

1. 以下の内容を **pom.xml** ファイルの **<build>** 要素に追加します。最新バージョンの Maven プラグインと、**org.jboss.eap:wildfly-galleon-pack** Galleon 機能パックの最新バージョンを指定する必要があります。以下に例を示します。

```
<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>
```



注記

この例では、**web-clustering** Galleon レイヤーを使用して Web セッション共有を有効にします。

2. 以下の設定を含む **src/main/webapp/WEB-INF** ディレクトリーに **web.xml** ファイルを作成します。

```
<?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/> タグは、このサーブレットを複数のサーバーに分散できることを示します。

3. java ファイルを保存するディレクトリーを作成します。

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

APPLICATION_ROOT は、アプリケーションの **pom.xml** 設定ファイルが含まれるディレクトリーです。

4. 以下の内容で Java ファイル **MyServlet.java** を作成し、ファイルを **APPLICATION_ROOT/src/main/java/com/example/webclustering/** ディレクトリーに保存します。

```
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 の内容は、クライアントが HTTP リクエストを送信するエンドポイントを定義します。

5. 以下の内容で Java ファイル **User.java** を作成し、ファイルを **APPLICATION_ROOT/src/main/java/com/example/webclustering/** ディレクトリーに保存します。

```
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. アプリケーションをパッケージ化します。

```
$ mvn package
```

7. **オプション**: JBoss EAP ベアメタルプラットフォームでアプリケーションを実行するには、Using a bootable JAR on a JBoss EAP bare-metal platform にある手順に従いますが、以下の違いがあります。

   a. JBoss EAP ベアメタルプラットフォームでは、以下の例のように、**java -jar** コマンドを使用して複数の起動可能な JAR インスタンスを実行できます。

   ```
   $ 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. **検証**: ノード 1 インスタンス (http://127.0.0.1:8080/clustering) でアプリケーションにアクセスできます。ユーザーセッション ID とユーザー相関時間を書き留めます。
   このインスタンスを強制終了した後に、ノード 2 インスタンス (http://127.0.0.1:8090/clustering) にアクセスできます。ユーザーは、セッション ID とノード 1 インスタンスのユーザー作成時間と一致する必要があります。

8. **オプション**: JBoss EAP OpenShift プラットフォームでアプリケーションを実行するには、Using a bootable JAR on a JBoss EAP OpenShift platform にある手順に従いますが、以下の手順を完了させてください。

   a. **<cloud/>** 要素をプラグイン設定に追加します。以下に例を示します。

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

   b. アプリケーションをリビルドします。

   ```
   $ mvn clean package
   ```

   c. **oc login** コマンドを使用して、OpenShift インスタンスにログインします。

   d. OpenShift で新しいプロジェクトを作成します。以下に例を示します。

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

   e. JBoss EAP OpenShift プラットフォームで web-clustering アプリケーションを実行するには、Pod が実行されているサービスアカウントに承認アクセスが付与される必要があります。サービスアカウントは Kubernetes REST API にアクセスできます。以下の例は、サービスアカウントに付与されている認可アクセスを示しています。

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

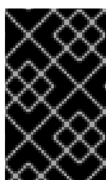
   f. 以下の **oc** コマンドを入力してアプリケーションイメージを作成します。

```
$ 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. **target/openshift** サブディレクトリーを作成します。パッケージ化されたアプリケーションは **openshift** サブディレクトリーにコピーされます。
2. 最新の OpenJDK 11 イメージストリームタグおよびイメージ情報を OpenShift プロジェクトにインポートします。
3. web-clustering ディレクトリーおよび OpenJDK 11 イメージストリームに基づいてビルド設定を作成します。
4. **target/openshift** サブディレクトリーをバイナリー入力として使用し、アプリケーションをビルドします。

g. アプリケーションのデプロイ:

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

$ oc expose svc/web-clustering
```



重要

現在の OpenShift namespace の他の Pod を表示するには **KUBERNETES_NAMESPACE** 環境変数を使用する必要があります。使用しない場合、サーバーは **default** 名前空間から Pod の取得を試行します。

h. ルートの URL を取得します。

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

i. 直前のコマンドから返された URL を使用して、Web ブラウザーでアプリケーションにアクセスします。以下に例を示します。

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

ユーザーセッション ID およびユーザー作成時間を書き留めます。

j. アプリケーションを 2 つの Pod にスケーリングします。

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

k. 以下のコマンドを実行して、利用可能な OpenShift Pod のリストを表示し、Pod のビルドステータスを確認します。

```
$ oc get pods
```

l. **oc delete pod web-clustering-POD_NAME** コマンドを使用して最も古い Pod を強制終了します。**POD_NAME** は最も古い Pod の名前です。

m. アプリケーションを再度アクセスします。

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

想定される結果: 新規 Pod で生成されるセッション ID および作成時間は、終了した Pod のものに一致します。これは、Web セッションデータストレージが有効になっていることを示します。

関連情報

- 分散可能な Web セッション管理プロファイルの詳細は、**開発ガイド**の 分散可能な Web セッション設定の distributable-web サブシステム を参照してください。
- JGroups プロトコルスタックの設定の詳細は、**JBoss EAP for OpenShift Container Platform のスタートガイド**の JGroups 検出メカニズムの設定 を参照してください。

8.15. CLI スクリプトを使用した起動可能な JAR の HTTP 認証の有効化

CLI スクリプトを使用して、起動可能な JAR の HTTP 認証を有効にできます。このスクリプトは、セキュリティーレルムとセキュリティードメインをサーバーに追加します。

前提条件

- **MAVEN_PLUGIN_VERSION.X.GA.Final-redhat-00001** などの最新の Maven プラグインバージョンを確認している。**MAVEN_PLUGIN_VERSION** はメジャーバージョンで、**X** はマイクロバージョンです。/ga/org/wildfly/plugins/wildfly-jar-maven-plugin のインデックス を参照してください。
- **4.0.X.GA-redhat-BUILD_NUMBER** などの最新の Galleon feature-pack バージョンを確認している。**X** は JBoss EAP XP のマイクロバージョンで、**BUILD_NUMBER** は Galleon 機能パックのビルド番号。JBoss EAP XP 4.0.0 製品のライフサイクル中に、**X** と **BUILD_NUMBER** の両方を進化できる。Index of /ga/org/jboss/eap/wildfly-galleon-pack のインデックス を参照してください。
- Maven プロジェクトを作成し、親依存関係を設定して、HTTP 認証を必要とするアプリケーションを作成するための依存関係を追加している。Creating a bootable JAR Maven project を参照してください。

重要

Maven プロジェクトを設定する場合は、Maven archetype 設定で HTTP 認証値を指定する必要があります。以下に例を示します。

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

注記

この手順の例では、以下のプロパティーを指定します。

- Maven プラグインバージョンの場合は、**${bootable.jar.maven.plugin.version}** です。
- Galleon 機能パックバージョンの場合は、**${jboss.xp.galleon.feature.pack.version}** です。

これらのプロパティーをプロジェクトで設定する必要があります。以下に例を示します。

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

手順

1. 以下の内容を **pom.xml** ファイルの **<build>** 要素に追加します。最新バージョンの Maven プラグインと、**org.jboss.eap:wildfly-galleon-pack** Galleon 機能パックの最新バージョンを指定する必要があります。以下に例を示します。

```
<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>
```

この例には、**elytron** サブシステムが含まれる **datasources-web-server** Galleon レイヤーが含まれていました。

2. **src/main/webapp/WEB-INF** ディレクトリーの **web.xml** ファイルを更新します。以下に例を示します。

```
<?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. java ファイルを保存するディレクトリーを作成します。

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

APPLICATION_ROOT は Maven プロジェクトのルートディレクトリーです。

4. 以下の内容で Java ファイル **TestServlet.java** を作成し、ファイルを **APPLICATION_ROOT/src/main/java/com/example/authentication/** ディレクトリーに保存します。

```
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();
    }

}
```

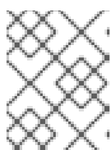
5. **authentication.cli** などの CLI スクリプトを作成し、これを **APPLICATION_ROOT/scripts** ディレクトリーなどの起動可能な JAR のアクセス可能なディレクトリーに保存します。スクリプトには以下のコマンドが含まれている必要があります。

```
/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. プラグイン **<configuration>** 要素に以下の設定抽出を追加します。

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

この例は、サーバーに定義されたセキュリティードメインにデフォルトの **undertow** セキュリティードメインを設定する **authentication.cli** CLI スクリプトを示しています。



注記

パッケージ化時にではなく、実行時に CLI スクリプトを実行するオプションがあります。これを行うには、この手順をスキップして手順 10 に進みます。

7. Maven プロジェクトのルートディレクトリーで、JBoss EAP JAR Maven プラグインが起動可能な JAR に追加するプロパティーファイルを保存するディレクトリーを作成します。

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

APPLICATION_ROOT は、アプリケーションの **pom.xml** 設定ファイルが含まれるディレクトリーです。

このディレクトリーには、**bootable-users.properties** および **bootable-groups.properties** などのファイルを保存します。

bootable-users.properties ファイルには以下の内容が含まれます。

```
testuser=bootable_password
```

bootable-groups.properties ファイルには以下の内容が含まれます。

```
testuser=Users
```

8. 以下の **extra-content-content-dirs** 要素を既存の **<configuration>** 要素に追加します。

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

extra-content ディレクトリーには、プロパティーファイルが含まれます。

9. アプリケーションを起動可能な JAR としてパッケージ化します。

```
$ mvn package
```

10. アプリケーションを起動します。

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

手順 6 をスキップし、ビルド中に CLI スクリプトを実行しないことを選択した場合は、次のコマンドを使用してアプリケーションを起動します。

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

11. サーブレットを呼び出しますが、認証情報は指定しないでください。

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

想定される出力:

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

12. サーバーを呼び出して認証情報を指定します。以下に例を示します。

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

起動可能な JAR に対して HTTP 認証が有効になっていることを示す HTTP 200 ステータスが返されます。以下に例を示します。

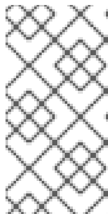
```
HTTP/1.1 200 OK
....
Hello testuser
```

関連情報

- **undertow** セキュリティードメインの HTTP 認証を有効にする方法は、**サーバーセキュリティーの設定方法**の CLI セキュリティーコマンドを使用したアプリケーションの HTTP 認証の有効化 を参照してください。

8.16. RED HAT SINGLE SIGN-ON での JBOSS EAP の起動可能な JAR アプリケーションのセキュア化

Galleon **keycloak-client-oidc** レイヤーを使用して、Red Hat Single Sign-On 7.5 OpenID Connect クライアントアダプターでプロビジョニングされたバージョンのサーバーをインストールできます。



注記

keycloak-client-oidc レイヤーの使用は、JBoss EAP XP 4 では非推奨になりました。代わりに、ネイティブの OpenID Connect (OIDC) クライアントを提供する **elytron-oidc-client** レイヤーを使用してください。詳細については、Developing JBoss EAP bootable jar application with OpenID Connect を参照してください。

keycloak-client-oidc レイヤーは、Red Hat Single Sign-On OpenID Connect クライアントアダプターを Maven プロジェクトに提供します。このレイヤーは、**keycloak-adapter-galleon-pack** Red Hat Single Sign-On 機能パックに含まれています。

keycloak-adapter-galleon-pack 機能パックを JBoss EAP Maven プラグイン設定に追加し、**keycloak-client-oidc** を追加できます。Supported Configurations: Red Hat Single Sign-On 7.4 ページにアクセスすると、JBoss EAP と互換性のある Red Hat Single Sign-On クライアントアダプターを表示できます。

この手順の例では、**keycloak-client-oidc** レイヤーによって提供される JBoss EAP の機能を使用し、JBoss EAP の起動可能な JAR をセキュアにする方法を説明します。

前提条件

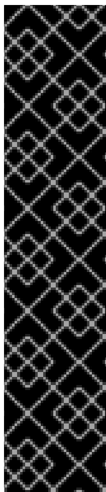
- **MAVEN_PLUGIN_VERSION.X.GA.Final-redhat-00001** などの最新の Maven プラグインバージョンを確認している。**MAVEN_PLUGIN_VERSION** はメジャーバージョンで、**X** はマイクロバージョンです。/ga/org/wildfly/plugins/wildfly-jar-maven-plugin のインデックス を参照してください。

- **4.0.X.GA-redhat-BUILD_NUMBER** などの最新の Galleon feature-pack バージョンを確認している。**X** は JBoss EAP XP のマイクロバージョンで、**BUILD_NUMBER** は Galleon 機能パックのビルド番号。JBoss EAP XP 4.0.0 製品のライフサイクル中に、**X** と **BUILD_NUMBER** の両方を進化できる。Index of /ga/org/jboss/eap/wildfly-galleon-pack のインデックス を参照してください。

- **org.keycloak:keycloak-adapter-galleon-pack:15.0.X.redhat-BUILD_NUMBER** などの最新の Red Hat Single Sign-On Galleon 機能パックバージョンを確認している。**X** は、アプリケーションのセキュア化に使用する Red Hat Single Sign-On のマイクロバージョンである Red Hat Single Sign-On のマイクロバージョンです。**BUILD_NUMBER** は Red Hat Single Sign-On Galleon 機能パックのビルド番号です。JBoss EAP XP 4.0.0 製品のライフサイクル中に、**X** と **BUILD_NUMBER** の両方を進化できる。Index of /ga/org/keycloak/keycloak-adapter-galleon-pack を参照してください。

- Red Hat Single Sign-On でセキュア化するアプリケーションを作成するために、Maven プロジェクトを作成し、親依存関係を設定して依存関係を追加している。Creating a bootable JAR Maven project を参照してください。

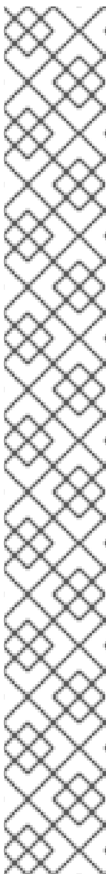
- 8090 ポートで実行している Red Hat Single Sign-On サーバーがある。Red Hat Single Sign-On サーバーの起動 を参照してください。
- Red Hat Single Sign-On 管理コンソールにログインし、以下のメタデータを作成している。
  - **demo** という名前のレルム。
  - **Users** という名前のロール。
  - ユーザーおよびパスワード**Users** ロールをユーザーに割り当てる必要があります。
  - ルート URL を含むパブリッククライアントの **Web** アプリケーション。この手順の例では、web アプリケーションおよび Root URL **http://localhost:8080/simple-webapp/secured** として simple-webapp を定義します。



重要

Maven プロジェクトを設定する場合は、Maven archetype で Red Hat Single Sign-On でセキュリティー保護するアプリケーションの値を指定する必要があります。以下に例を示します。

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



注記

この手順の例では、以下のプロパティーを指定します。

- Maven プラグインバージョンの場合は、**${bootable.jar.maven.plugin.version}** です。
- Galleon 機能パックバージョンの場合は、**${jboss.xp.galleon.feature.pack.version}** です。
- Red Hat Single Sign-On 機能パックバージョンの場合は、**${keycloak.feature.pack.version}** です。

これらのプロパティーをプロジェクトで設定する必要があります。以下に例を示します。

```
<properties>
    <bootable.jar.maven.plugin.version>6.1.2.Final-redhat-00001</bootable.jar.maven.plugin.version>
    <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-00002</jboss.xp.galleon.feature.pack.version>
    <keycloak.feature.pack.version>15.0.4.redhat-00001</keycloak.feature.pack.version>
</properties>
```

手順

1. 以下の内容を **pom.xml** ファイルの **<build>** 要素に追加します。最新バージョンの Maven プラグインと、**org.jboss.eap:wildfly-galleon-pack** Galleon 機能パックの最新バージョンを指定する必要があります。以下に例を示します。

```
<plugins>
    <plugin>
        <groupId>org.wildfly.plugins</groupId>
        <artifactId>wildfly-jar-maven-plugin</artifactId>
        <version>${bootable.jar.maven.plugin.version}</version>
        <configuration>
            <feature-packs>
                <feature-pack>
                    <location>org.jboss.eap:wildfly-galleon-
pack:${jboss.xp.galleon.feature.pack.version}</location>
                </feature-pack>
                <feature-pack>
                    <location>org.keycloak:keycloak-adapter-galleon-
pack:${keycloak.feature.pack.version}</location>
                </feature-pack>
            </feature-packs>
            <layers>
                <layer>datasources-web-server</layer>
                <layer>keycloak-client-oidc</layer>
            </layers>
        </configuration>
        <executions>
            <execution>
                <goals>
                    <goal>package</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
```

Maven プラグインは、Web アプリケーションのデプロイに必要なサブシステムとモジュールをプロビジョニングします。

keycloak-client-oidc レイヤーは、**keycloak** サブシステムとその依存関係を使用して Red Hat Single Sign-On 認証のサポートをアクティベートすることで、Red Hat Single Sign-On の OpenID Connect クライアントアダプターをプロジェクトに提供します。Red Hat Single Sign-On クライアントアダプターは、Red Hat Single Sign-On でアプリケーションとサービスのセキュリティーを保護するライブラリーです。

2. **pom.xml** ファイルでは、プラグイン設定で **<context-root>** を **false** に設定します。これにより、**simple-webapp** リソースパスにアプリケーションが登録されます。デフォルトでは、WAR ファイルは root-context パスで登録されます。

```
<configuration>
    ...
     <context-root>false</context-root>
     ...
</configuration>
```

3. **configure-oidc.cli** などの CLI スクリプトを作成し、**APPLICATION_ROOT/scripts** ディレクトリーなどの起動可能な JAR のアクセス可能なディレクトリーに保存しま

す。**APPLICATION_ROOT** は Maven プロジェクトのルートディレクトリーです。スクリプトには、以下の例のようなコマンドを含める必要があります。

```
/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)
```

このスクリプトの例では、**keycloak** サブシステムで **secure-deployment=simple-webapp.war** リソースを定義します。**simple-webapp.war** リソースは、起動可能な JAR にデプロイされる WAR ファイルの名前です。

4. プロジェクトの **pom.xml** ファイルで、以下の設定抽出を既存のプラグイン **<configuration>** 要素に追加します。

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

5. **src/main/webapp/WEB-INF** ディレクトリーの **web.xml** ファイルを更新します。以下に例を示します。

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

<web-app version="4.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_4_0.xsd"
  metadata-complete="false">

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

</web-app>
```

6. **オプション:** ステップ 7 から 9 の代わりに、 **keycloak.json** 記述子を web アプリケーションの **WEB-INF** ディレクトリーに追加して、Web アプリケーションにサーバー設定を埋め込むことも可能です。以下に例を示します。

```
{
    "realm" : "demo",
    "resource" : "simple-webapp",
    "public-client" : "true",
    "auth-server-url" : "http://localhost:8090/auth/",
    "ssl-required" : "EXTERNAL"
}
```

次に、Web アプリケーションの **<auth-method>** を **KEYCLOAK** に設定する必要があります。以下のコード例は、**<auth-method>** を設定する方法を示しています。

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

7. 以下の内容で **SecuredServlet.java** という名前の Java ファイルを作成し、ファイルを **APPLICATION_ROOT/src/main/java/com/example/securedservlet/** ディレクトリーに保存します。

```
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. アプリケーションを起動可能な JAR としてパッケージ化します。

```
$ mvn package
```

9. アプリケーションを起動します。以下の例では、指定された起動可能な JAR パスから **simple-webapp** Web アプリケーションを起動します。

```
$ java -jar target/simple-webapp-bootable.jar
```

10. Web ブラウザーで以下の URL を指定して、Red Hat Single Sign-On でセキュア化された Web ページにアクセスします。以下の例は、セキュアな **simple-webapp** Web アプリケーションの URL を示しています。

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

11. Red Hat Single Sign-On レルムからユーザーとしてログインします。

12. **検証:** Web ページに以下の出力が表示されることを確認します。

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

関連情報

- Red Hat Single Sign-On アダプターサブシステムの設定に関する情報は、**Securing Applications and Services Guide** の JBoss EAP Adapter を参照してください。
- プロジェクトに JBoss EAP JAR Maven を指定する方法は、Specifying Galleon layers for your bootable JAR server を参照してください。

8.17. DEV モードでの起動可能な JAR のパッケージ化

JBoss EAP JAR Maven プラグインの **dev goal** は、アプリケーションの開発プロセスを強化する **dev** モードである Development Mode を提供します。

dev モードでは、アプリケーションに変更を加えた後、起動可能な JAR をリビルドする必要はありません。

この手順のワークフローは、**dev** モードを使用して起動可能な JAR を設定する方法を示しています。

前提条件

- Maven がインストールされている。
- Maven プロジェクトを作成し、親依存関係を設定して、MicroProfile アプリケーションを作成するための依存関係を追加している。MicroProfile Config development を参照してください。
- Maven プロジェクトの **pom.xml** ファイルに JBoss EAP JAR Maven plug-in を指定しました。

手順

1. 開発モードで起動可能な JAR をビルドして起動します。

```
$ mvn wildfly-jar:dev
```

dev モードでは、サーバーデプロイメントスキャナーは **target/deployments** ディレクトリーを監視するように設定されています。

2. 以下のコマンドで、アプリケーションを **target/deployments** ディレクトリーにビルドしてコピーするよう JBoss EAP Maven プラグインに指示します。

```
$ mvn package -Ddev
```

起動可能な JAR 内にパッケージ化されたサーバーは、**target/deployments** ディレクトリーに保存されているアプリケーションをデプロイします。

3. アプリケーションコードのコードを変更します。

4. **mvn package -Ddev** を使用して、JBoss EAP Maven プラグインにアプリケーションを再ビルドして再デプロイするように指示します。

5. サーバーの停止以下に例を示します。

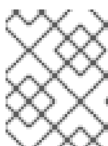
```
$ mvn wildfly-jar:shutdown
```

6. アプリケーションの変更が完了したら、アプリケーションを起動可能な JAR としてパッケージ化します。

```
$ mvn package
```

8.18. サーバーアーティファクトのアップグレード

サーバーアーティファクトは、JBoss Modules モジュール内にある jar ファイルであり、プロジェクト pom.xml ファイルの Maven コーディネートを使用して参照できます。



注記

サーバーアーティファクトをアップグレードすると、サポートされていない設定になる可能性があることに注意してください。

前提条件

- Maven アーティファクトがローカル Maven リポジトリーまたはリモート Maven リポジトリーからアクセス可能であることを確認してください。

手順

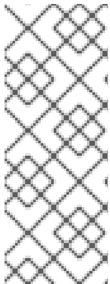
1. サーバーアーティファクトを正常にアップグレードするには、ビルド中に依存関係に存在するアーティファクトのバージョンを使用します。以下に例を示します。

```
<dependencies>
  ...
  <dependency>
 <groupId>io.undertow</groupId>
 <artifactId>undertow-core</groupId>
 <version>2.2.5.Final-redhat-00001</version>
       <scope>provided</scope>
       <!-- In order to avoid bringing transitive dependencies to the project, exclude all
dependencies -->
       <exclusions>
         <exclusion>
           <groupId>*</groupId>
```

```
        <artifactId>*</artifactId>
       </exclusion>
      </exclusions>
   </dependency>
   ...
 </dependencies>
```

2. プラグインの **<configuration>** セクションを開き、**<overridden-server-artifacts>** リスト内のアーティファクト groupId と artifactId を更新します。次に例を示します。

```
<configuration>
 ...
 <overridden-server-artifacts>
  <artifact>
   <groupId>io.undertow</groupId>
   <artifactId>undertow-core</groupId>
  </artifact>
 </overridden-server-artifacts>
</configuration>
```

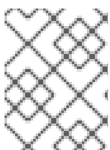


注記

- **<overridden-server-artifacts>** に追加されたアーティファクトがプロジェクトの依存関係の中に見つからない場合、失敗となります。
- **<overridden-server-artifacts>**` に追加されたアーティファクトがプロビジョニングされたサーバーアーティファクトの中にない場合、アップグレード対象のアーティファクトを見つけることができないため失敗となります。

8.19. EAP7.4.GA 依存関係の更新

JBoss EAP XP 4.0.0 の起動可能な JAR をビルドする場合、JBoss EAP XP 4.0.0 の JBoss EAP 7.4 への依存関係を更新できます。JBoss EAP XP4.0.0 galleon 機能パック **org.jboss.eap:wildfly-galleon-pack:4.0.0.GA-redhat-00001** は、**org.jboss.eap:wildfly-ee-galleon-pack:7.4.0.GA-redhat-00001** に依存しており、ブート可能な JAR をビルドするときにアップグレードできます。



注記

最新の JBoss EAP XP バージョンにアップグレードします。これにより、JBoss EAP XP 4.0.0 の起動可能な JAR で最新の更新を確実に取得できます。

前提条件

- 最新バージョンの JBoss EAP XP があります。

手順

1. JBoss EAP Galleon 機能パック Maven アーティファクトがローカルまたはリモートの Maven リポジトリーからアクセス可能であることを確認します。
2. プロジェクトの依存関係に Galleon 機能パックアーティファクトを追加します。
   a. スコープを **provided** に設定します。

b. タイプを **zip** に設定します。

c. アーティファクトのバージョンを設定します。以下に例を示します。

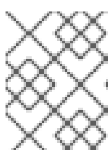
```
<dependencies>
  …
  <dependency>
 <groupId>org.jboss.eap</groupId>
 <artifactId>wildfly-ee-galleon-pack</groupId>
  <version>7.4.1.GA-redhat-00001</version>
<scope>provided</scope>
<type>zip</type>
  </dependency>
  ...
</dependencies>
```

3. プラグインの **<configuration>** セクションを開き、**<overridden-server-artifacts>** リスト内のアーティファクト groupId と artifactId を更新します。次に例を示します。

```
<configuration>
 …
 <overridden-server-artifacts>
  <artifact>
   <groupId>org.jboss.eap</groupId>
   <artifactId>wildfly-ee-galleon-pack</groupId>
  </artifact>
 </overridden-server-artifacts>
</configuration>
```

4. ビルド中に最新バージョンの JBoss EAP XP Galleon 機能パックを使用して、依存関係を正常に設定します。

8.20. 起動可能な JAR への JBOSS EAP パッチの適用

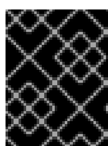


注記

JBoss EAP XP 4.0.0 では、起動可能な jar のレガシーパッチ機能が非推奨になりました。

JBoss EAP ベアメタルプラットフォームでは、CLI スクリプトを使用して起動可能な JAR にパッチをインストールできます。

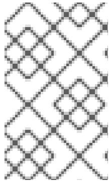
CLI スクリプトは **patch apply** コマンドを実行し、起動可能な JAR ビルド時にパッチを適用します。



重要

起動可能な JAR にパッチを適用した後は、適用されたパッチからロールバックすることはできません。パッチなしで起動可能な JAR をリビルドする必要があります。

さらに、JBoss EAP JAR Maven プラグインを使用して、起動可能な JAR にレガシーパッチを適用することもできます。このプラグインは、サーバーのパッチに使用される CLI スクリプトを参照する **<legacy-patch-cli-script>** 設定オプションを提供します。



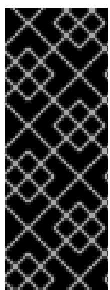
注記

<legacy-patch-cli-script> の接頭辞 **legacy-*** は、アーカイブパッチを起動可能な JAR に適用することに関連します。このメソッドは、通常の JBoss EAP ディストリビューションにパッチを適用するのと似ています。

JBoss EAP JAR Maven プラグイン設定で **legacy-patch-cleanup** オプションを使用して、未使用のパッチコンテンツを削除して、起動可能な JAR のメモリーフットプリントを低減できます。このオプションは、未使用のモジュール依存関係を削除します。このオプションは、パッチ設定ファイルで、デフォルトで **false** に設定されています。

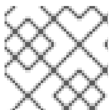
legacy-patch-cleanup オプションは、以下のパッチコンテンツを削除します。

- **<JBOSS_HOME>/.installation/patches** ディレクトリー
- ベースレイヤーのパッチモジュールの元の場所。
- パッチによって追加された未使用のモジュールは、既存のモジュールグラフまたはパッチが適用されたモジュールグラフで参照されません。
- **.overlays** ファイルに記載されていないディレクトリーのオーバーレイを設定します。



重要

legacy-patch-clean-up オプション変数は、テクノロジープレビューとして提供されます。テクノロジープレビューの機能は、Red Hat の本番環境のサービスレベルアグリーメント (SLA) ではサポートされず、機能的に完全ではないことがあるため、Red Hat は本番環境での使用は推奨しません。テクノロジープレビューの機能は、最新の技術をいち早く提供して、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。



注記

この手順に概説する情報は、起動可能な JAR にも関係します。

前提条件

- Red Hat カスタマーポータル でアカウントを設定している。
- **製品のダウンロード**ページから以下のファイルをダウンロードしている。
  - JBoss EAP 7.4.4 GA パッチ
  - JBoss EAP XP 4.0.0 パッチ

手順

1. 起動可能な JAR に適用するレガシーパッチを定義する CLI スクリプトを作成します。スクリプトには、1 つまたは複数の patch apply コマンドが含まれている必要があります。Galleon レイヤーでトリミングされたサーバーにパッチを適用する場合には、**--override-all** コマンドが必要です。以下に例を示します。

```
patch apply patch-oneoff1.zip --override-all

patch apply patch-oneoff2.zip --override-all
```

```
patch info --json-output
```

2. **pom.xml** ファイルの **<legacy-patch-cli-script>** 要素で CLI スクリプトを参照します。

3. 起動可能な JAR をリビルドします。

関連情報

- JBoss EAP MicroProfile Maven リポジトリーのダウンロードに関する詳細は、Downloading the JBoss EAP MicroProfile Maven repository patch as an archive file を参照してください。

- CLI スクリプトの作成に関する詳細は、CLI Scripts を参照してください。

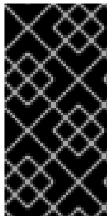
- テクノロジープレビュー機能の情報は、**Red Hat カスタマーポータル**の テクノロジープレビュー機能のサポート範囲 を参照してください。

第9章 JBOSS EAP での OPENID CONNECT

JBoss EAP ネイティブ OpenID Connect (OIDC) クライアントを使用して、外部 OpenID プロバイダーを介してアプリケーションをセキュリティー保護します。OIDC は、JBoss EAP などのクライアントが OpenID プロバイダー認証に基づいてユーザーの ID を検証できるようにする ID レイヤーです。たとえば、OpenID プロバイダーとして Red Hat Single Sign-On を使用して、JBoss EAP アプリケーションをセキュリティー保護できます。

9.1. JBOSS EAP での OPENID CONNECT 設定

OpenID プロバイダーを使用してアプリケーションを保護する場合、セキュリティードメインリソースをローカルで設定する必要はありません。**elytron-oidc-client** サブシステムは、OpenID プロバイダーと接続するための JBoss EAP のネイティブ OpenID Connect (OIDC) クライアントを提供します。JBoss EAP は、OpenID プロバイダーの設定に基づいて、アプリケーションの仮想セキュリティードメインを自動的に作成します。

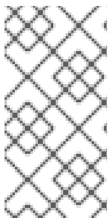


重要

Red Hat Single Sign-On で OIDC クライアントを使用することを推奨します。JSON Web トークン (JWT) であるアクセストークンを使用するように設定でき、RS256、RS384、RS512、ES256、ES384、または ES512 署名アルゴリズムを使用するように設定できる場合は、他の OpenID プロバイダーを使用できます。

OIDC の使用を有効にするには、**elytron-oidc-client** サブシステムまたはアプリケーション自体を設定できます。JBoss EAP は、次のように OIDC 認証をアクティブにします。

- アプリケーションを JBoss EAP にデプロイすると、**elytron-oidc-client** サブシステムがデプロイメントをスキャンして、OIDC 認証メカニズムが必要かどうかを検出します。
- サブシステムが **elytron-oidc-client** サブシステムまたはアプリケーションデプロイメント記述子のいずれかでデプロイメントの OIDC 設定を検出した場合、JBoss EAP はアプリケーションの OIDC 認証メカニズムを有効にします。
- サブシステムが両方の場所で OIDC 設定を検出した場合、**elytron-oidc-client** サブシステム **secure-deployment** 属性の設定が、アプリケーションデプロイメント記述子の設定よりも優先されます。



注記

Red Hat Single Sign-On でアプリケーションをセキュリティー保護するための **keycloak-client-oidc** レイヤーは、JBoss EAPXP4.0.0 では非推奨になっています。代わりに、**elytron-oidc-client** サブシステムによって提供されるネイティブ OIDC クライアントを使用してください。

デプロイメント設定

デプロイメント記述子を使用して OIDC でアプリケーションを保護するには、アプリケーションのデプロイメント設定を次のように更新します。

- OIDC 設定情報を含む **oidc.json** というファイルを **WEB-INF** ディレクトリーに作成します。

oidc.json コンテンツの例

```
{
```

```
  "client-id" : "customer-portal", (1)
  "provider-url" : "http://localhost:8180/auth/realms/demo", (2)
  "ssl-required" : "external", (3)
   "credentials" : {
     "secret" : "234234-234234-234234" (4)
   }
}
```

1. OpenID プロバイダーで OIDC クライアントを識別するための名前。
2. OpenID プロバイダーの URL。
3. 外部リクエストには HTTPS が必要です。
4. OpenID プロバイダーに登録されたクライアントシークレット。

- アプリケーションデプロイメント記述子 **web.xml** ファイルで **auth-method** プロパティーを **OIDC** に設定します。

デプロイメント記述子の更新例

```
<login-config>
    <auth-method>OIDC</auth-method>
</login-config>
```

サブシステムの設定

次の方法で **elytron-oidc-client** サブシステムを設定することで、OIDC を使用してアプリケーションを保護できます。

- アプリケーションごとに同じ OpenID プロバイダーを使用する場合は、複数のデプロイメントに対して単一の設定を作成します。
- アプリケーションごとに異なる OpenID プロバイダーを使用する場合は、デプロイメントごとに異なる設定を作成します。

単一デプロイメントの XML 設定の例:

```
<subsystem xmlns="urn:wildfly:elytron-oidc-client:1.0">
    <secure-deployment name="DEPLOYMENT_RUNTIME_NAME.war"> (1)
        <client-id>customer-portal</client-id> (2)
        <provider-url>http://localhost:8180/auth/realms/demo</provider-url> (3)
        <ssl-required>external</ssl-required> (4)
        <credential name="secret" secret="0aa31d98-e0aa-404c-b6e0-e771dba1e798" /> (5)
    </secure-deployment
</subsystem>
```

1. デプロイメントランタイム名。
2. OpenID プロバイダーで OIDC クライアントを識別するための名前。
3. OpenID プロバイダーの URL。
4. 外部リクエストには HTTPS が必要です。

5 OpenID プロバイダーに登録されたクライアントシークレット。

同じ OpenID プロバイダーを使用して複数のアプリケーションを保護するには、次の例に示すように、**provider** を個別に設定します。

```
<subsystem xmlns="urn:wildfly:elytron-oidc-client:1.0">
    <provider name="${OpenID_provider_name}">
        <provider-url>http://localhost:8080/auth/realms/demo</provider-url>
        <ssl-required>external</ssl-required>
    </provider>
    <secure-deployment name="customer-portal.war"> 1
        <provider>${OpenID_provider_name}</provider>
        <client-id>customer-portal</client-id>
        <credential name="secret" secret="0aa31d98-e0aa-404c-b6e0-e771dba1e798" />
    </secure-deployment>
    <secure-deployment name="product-portal.war"> 2
        <provider>${OpenID_provider_name}</provider>
        <client-id>product-portal</client-id>
        <credential name="secret" secret="0aa31d98-e0aa-404c-b6e0-e771dba1e798" />
    </secure-deployment>
</subsystem>
```

1 デプロイメント: **customer-portal.war**

2 別のデプロイメント: **product-portal.war**

関連情報

- OpenID Connect specification
- **elytron-oidc-client** サブシステム属性
- OpenID Connect Libraries
- Securing applications using OpenID Connect with Red Hat Single Sign-On
- MicroProfile JWT

9.2. ELYTRON-OIDC-CLIENT サブシステムの有効化

elytron-oidc-client サブシステムは、**standalone-microprofile.xml** 設定ファイルで提供されます。これを使用するには、**bin/standalone.sh -c standalone-microprofile.xml** コマンドを使用してサーバーを起動する必要があります。管理 CLI を使用して有効にすることにより、**elytron-oidc-client** サブシステムを **standalone.xml** 設定に含めることができます。

前提条件

- JBoss EAP XP がインストールされている。

手順

1. 管理 CLI を使用して **elytron-oidc-client** エクステンションを追加します。

```
/extension=org.wildfly.extension.elytron-oidc-client:add
```

2. 管理 CLI を使用して **elytron-oidc-client** サブシステムを有効にします。

```
/subsystem=elytron-oidc-client:add
```

3. Reload JBoss EAP.

```
reload
```

これで、コマンド **bin/standalone.sh** を使用してサーバーを通常どおりに起動することにより、**elytron-oidc-client** サブシステムを使用できます。

関連情報

- **elytron-oidc-client** subsystem attributes

9.3. SECURING APPLICATIONS USING OPENID CONNECT WITH RED HAT SINGLE SIGN-ON

OpenID Connect (OIDC) を使用して、認証を外部の OpenID プロバイダーに委任できます。**elytron-oidc-client** サブシステムは、外部の OpenID プロバイダーと接続するための JBoss EAP のネイティブ OIDC クライアントを提供します。

Red Hat Single Sign-On を使用して OpenID Connect でセキュリティー保護されたアプリケーションを作成するには、次の手順に従います。

- OpenID プロバイダーとして Red Hat Single Sign-On を設定する
- アプリケーションの Maven プロジェクトを作成する
- OpenID Connect を使用するアプリケーションを作成する
- ユーザーのロールに基づいてアプリケーションへのアクセスを制限する
- Red Hat Single Sign-On でユーザーロールを作成して割り当てる

9.3.1. OpenID プロバイダーとしての Red Hat Single Sign-On の設定

Red Hat Single Sign-On は、Single Sign-On (SSO) で Web アプリケーションをセキュリティー保護するための ID およびアクセス管理プロバイダーです。OpenID Connect (OAuth 2.0 のエクステンション) をサポートしています。

前提条件

- Red Hat Single Sign-On サーバーをインストールしました。詳細については、Red Hat Single Sign-On **Getting Started Guide** の Installing the Red Hat Single Sign-On server を参照してください。
- Red Hat Single Sign-On サーバーインスタンスにユーザーを作成しました。詳細については、Red Hat Single Sign-On **Getting Started Guide** の Creating a user を参照してください。

手順

1. JBoss EAP のデフォルトポートは 8080 であるため、Red Hat Single Sign-On サーバーを 8080 以外のポートで起動します。

   構文

   ```
   $ RH_SSO_HOME/bin/standalone.sh -Djboss.socket.binding.port-offset=<offset-number>
   ```

   例:

   ```
   $ /home/servers/rh-sso-7.4/bin/standalone.sh -Djboss.socket.binding.port-offset=100
   ```

2. **http://localhost:<port>/auth/** で管理コンソールにログインします。たとえば、**http://localhost:8180/auth/** です。
3. レルムを作成するには、管理コンソールで **Master** にカーソルを合わせ、**Add realm** をクリックします。
4. レルムの名前を入力します。たとえば、**example_realm**。**Enabled** が **オン** になっていることを確認し、**Create** をクリックします。
5. **Users** をクリックし、ユーザーの **Add user** をクリックしてユーザーをレルムに追加します。
6. ユーザー名を入力。たとえば、**jane_doe**。User **Enabled** が **ON** になっていることを確認し、**Save** をクリックします。
7. **Credentials** をクリックして、ユーザーにパスワードを追加します。
8. ユーザーのパスワードを設定します。たとえば、**janedoep@$$**。**Temporary** を **OFF** に切り替えて、**Set Password** をクリックします。確認プロンプトで、**Set password** をクリックします。
9. **Clients** をクリックし、**Create** をクリックしてクライアント接続を設定します。
10. クライアント ID を入力します。たとえば、**my_jbeap**。**Client Protocol** が **openid-connect** に設定されていることを確認し、**Save** をクリックします。
11. **Installation** をクリックし、**Format Option** として **Keycloak OIDC JSON** を選択して、接続パラメーターを確認します。

    ```
    {
      "realm": "example_realm",
      "auth-server-url": "http://localhost:8180/auth/",
      "ssl-required": "external",
      "resource": "my_jbeap",
      "public-client": true,
      "confidential-port": 0
    }
    ```

    Red Hat Single Sign-On をアイデンティティプロバイダーとして使用するように JBoss EAP アプリケーションを設定する場合は、次のようにパラメーターを使用します。

    ```
    "provider-url" : "http://localhost:8180/auth/realms/example_realm",
    "ssl-required": "external",
    "client-id": "my_jbeap",
    ```

```
"public-client": true,
"confidential-port": 0
```

12. **Clients** をクリックし、**my_jbeap** の横にある **Edit** をクリックしてクライアント設定を編集します。

13. **Valid Redirect URIs** に、認証が成功した後にページがリダイレクトする URL を入力します。この例では、この値を **http://localhost:8080/simple-oidc-example/secured/*** に設定します

関連情報

- セキュアなアプリケーションを作成するための Maven プロジェクトの設定
- レルムおよびユーザーの作成

9.3.2. セキュアなアプリケーションを作成するための Maven プロジェクトの設定

セキュリティー保護されたアプリケーションを作成するために必要な依存関係とディレクトリー構造を使用して Maven プロジェクトを作成します。

前提条件

- Maven がインストールされている。詳細については、Downloading Apache Maven を参照してください。
- 最新リリース用に Maven リポジトリーを設定しました。詳細については、Maven and the JBoss EAP microprofile maven repository を参照してください。

手順

1. **mvn** コマンドを使用して Maven プロジェクトを設定します。このコマンドは、プロジェクトのディレクトリー構造と **pom.xml** 設定ファイルを作成します。

   構文

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

   例

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

2. アプリケーションのルートディレクトリーに移動します。

   構文

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

例

```
$ cd simple-oidc-example
```

3. 生成された **pom.xml** ファイルを次のように更新します。

   a. 以下のプロパティーを設定します。

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

   b. 次の依存関係を設定します。

```
<dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.1.0.redhat-1</version>
      <scope>provided</scope>
    </dependency>
</dependencies>
```

   c. **mvn widlfy:deploy** を使用してアプリケーションをデプロイするには、次のビルド設定を設定します。

```
<build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
        <plugin>
            <groupId>org.wildfly.plugins</groupId>
            <artifactId>wildfly-maven-plugin</artifactId>
            <version>2.1.0.Final</version>
        </plugin>
    </plugins>
</build>
```

検証

- アプリケーションのルートディレクトリーで、次のコマンドを入力します。

```
$ mvn install
```

次のような出力が得られます。

```
[INFO] ------------------------------------------------------------------------
```

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.440 s
[INFO] Finished at: 2021-12-27T14:45:12+05:30
[INFO] ------------------------------------------------------------------------
```

これで、セキュアなアプリケーションを作成できます。

関連情報

- OpenID Connect を使用するセキュアなアプリケーションの作成

9.3.3. OpenID Connect を使用するセキュアなアプリケーションの作成

アプリケーションをセキュリティー保護するには、デプロイメント設定を更新するか、**elytron-oidc-client** サブシステムを設定します。次の例は、ログインしたユーザーのプリンシパルを出力するサーブレットの作成を示しています。既存のアプリケーションの場合、デプロイメント設定または **elytron-oidc-client** サブシステムの更新に関連するステップのみが必要です。

この例では、プリンシパルの値は OpenID プロバイダーの ID トークンから取得されます。デフォルトでは、プリンシパルはトークンからの **"sub"** クレームの値です。次のいずれかで、ID トークンからプリンシパルとして使用するクレーム値を指定できます。

- **elytron-oidc-client** サブシステム属性 **principal-attribute**。
- **oidc.json** ファイル。

プロシージャの <application_root> は、**pom.xml** ファイルディレクトリーを示します。**pom.xml** ファイルには、アプリケーションの Maven 設定が含まれています。

前提条件

- Maven プロジェクトを作成しました。詳細は、Configuring Maven project for creating a secure application を参照してください。
- Red Hat Single Sign-On を OpenID プロバイダーとして設定しました。詳細は、Configuring Red Hat Single Sign-On as an OpenID provider を参照してください。
- **elytron-oidc-client** サブシステムを有効にしました。詳細は、Enabling the elytron-oidc-client subsystem を参照してください。

手順

1. Java ファイルを保存するディレクトリーを作成します。

   構文

   ```
   $ mkdir -p <application_root>/src/main/java/com/example/oidc
   ```

   例

   ```
   $ mkdir -p simple-oidc-example/src/main/java/com/example/oidc
   ```

2. 新しいディレクトリーに移動します。

構文

```
$ cd <application_root>/src/main/java/com/example/oidc
```

例

```
$ cd simple-oidc-example/src/main/java/com/example/oidc
```

3. 次の内容のサーブレット SecuredServlet.java を作成します。

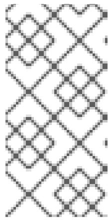
```
package com.example.oidc;

import java.io.IOException;
import java.io.PrintWriter;
import java.security.Principal;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * A simple secured HTTP servlet.
 *
 */
@WebServlet("/secured")
public class SecuredServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        try (PrintWriter writer = resp.getWriter()) {
            writer.println("<html>");
            writer.println("  <head><title>Secured Servlet</title></head>");
            writer.println("  <body>");
            writer.println("    <h1>Secured Servlet</h1>");
            writer.println("    <p>");
            writer.print(" Current Principal '");
            Principal user = req.getUserPrincipal();
            writer.print(user != null ? user.getName() : "NO AUTHENTICATED USER");
            writer.print("'");
            writer.println("    </p>");
            writer.println("  </body>");
            writer.println("</html>");
        }
    }

}
```

4. アプリケーションの **WEB-INF** ディレクトリーにあるデプロイメント記述子 **web.xml** ファイルに、アプリケーションにアクセスするためのセキュリティールールを追加します。

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

```
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  metadata-complete="false">

   <security-constraint>
      <web-resource-collection>
         <web-resource-name>secured</web-resource-name>
         <url-pattern>/secured</url-pattern>
      </web-resource-collection>
      <auth-constraint>
         <role-name>*</role-name>
      </auth-constraint>
   </security-constraint>

   <security-role>
      <role-name>*</role-name>
   </security-role>
</web-app>
```

5. OpenID Connect を使用してアプリケーションをセキュリティー保護するには、デプロイメント設定を更新するか、**elytron-oidc-client** サブシステムを設定します。



注記

デプロイメント設定と **elytron-oidc-client** サブシステムの両方で OpenID Connect を設定する場合、**elytron-oidc-client** サブシステムの **secure-deployment** 属性の設定は、アプリケーションデプロイメント記述子の設定よりも優先されます。

- デプロイメント設定の更新:

  i. 次のように、**WEB-INF** ディレクトリーにファイル **oidc.json** を作成します。

```
{
  "provider-url" : "http://localhost:8180/auth/realms/example_realm",
  "ssl-required": "external",
  "client-id": "my_jbeap",
  "public-client": true,
  "confidential-port": 0
}
```

  ii. デプロイメント記述子 **web.xml** ファイルを次のテキストで更新して、このアプリケーションが OIDC を使用することを宣言します。

```
<login-config>
   <auth-method>OIDC</auth-method>
</login-config>
```

- **elytron-oidc-client** サブシステムの設定:
  - アプリケーションを保護するには、次の管理 CLI コマンドを使用します。

```
/subsystem=elytron-oidc-client/secure-deployment=simple-oidc-example.war/:add(client-id=my_jbeap,provider-url=http://localhost:8180/auth/realms/example_realm,public-client=true,ssl-required=external)
```

6. アプリケーションのルートディレクトリーで、次のコマンドを使用してアプリケーションをコンパイルします。

```
$ mvn package
```

7. アプリケーションをデプロイします。

```
$ mvn wildfly:deploy
```

検証

1. ブラウザーで、**http://localhost:8080/simple-oidc-example/secured** に移動します。
2. クレデンシャルを使用してログインします。以下に例を示します。

```
username: jane_doe
password: janedoep@$$
```

次の出力が得られます。

```
Secured Servlet
Current Principal '5cb0c4ca-0477-44c3-bdef-04db04d7e39d'
```

これで、Red Hat Single Sign-On で OpenID プロバイダーとして設定したクレデンシャルを使用してアプリケーションにログインできます。

関連情報

- JBoss EAP での OpenID Connect 設定
- ユーザーのロールに基づいてアプリケーションへのアクセスを制限する

9.3.4. ユーザーのロールに基づいてアプリケーションへのアクセスを制限する

ユーザーのロールに基づいて、アプリケーションのすべてまたは一部へのアクセスを制限できます。たとえば、public ロールを持つユーザーに、機密性の低いアプリケーションの部分へのアクセスを許可し、admin ロールを持つユーザーに、機密性の高い部分へのアクセス権を与えることができます。

前提条件

- OpenID Connect を使用してアプリケーションをセキュリティー保護しました。詳細は、Creating a secure application that uses OpenID Connect を参照してください。

手順

1. デプロイメント記述子 **web.xml** ファイルを次のテキストで更新します。

構文

```
<security-constraint>
    ...
    <auth-constraint>
        <role-name><allowed_role></role-name>
    </auth-constraint>
</security-constraint>
```

例

```
<security-constraint>
    ...
    <auth-constraint>
        <role-name>example_role</role-name> (1)
    </auth-constraint>
</security-constraint>
```

(1) ロール **example_role** を持つユーザーのみにアプリケーションへのアクセスを許可します。

2. アプリケーションのルートディレクトリーで、次のコマンドを使用してアプリケーションを再コンパイルします。

```
$ mvn package
```

3. アプリケーションをデプロイします。

```
$ mvn wildfly:deploy
```

検証

1. ブラウザーで、**http://localhost:8080/simple-oidc-example/secured** に移動します。
2. クレデンシャルを使用してログインします。以下に例を示します。

```
username: jane_doe
password: janedoep@$$
```

次の出力が得られます。

```
Forbidden
```

ユーザー jane_doe に必要なロールを割り当てていないため、jane_doe はアプリケーションにログインできません。必要なロールを持つユーザーのみがログインできます。

ユーザーに必要なロールを割り当てるには、Creating and assigning roles to users in Red Hat Single Sign-On を参照してください。

9.3.5. Red Hat Single Sign-On でのユーザーロールの作成と割り当て

Red Hat Single Sign-On は、シングルサインオン (SSO) を使用して Web アプリケーションを保護するためのアイデンティティーおよびアクセス管理プロバイダーです。Red Hat Single Sign-On でユーザーを定義し、ロールを割り当てることができます。

前提条件

- Red Hat Single Sign-On を設定しました。詳細は、Configuring Red Hat Single Sign-On as an OpenID provider を参照してください。

手順

1. http://localhost:<port>/auth/ で管理コンソールにログインします。たとえば、http://localhost:8180/auth/ です。
2. JBoss EAP との接続に使用するレルムをクリックします。たとえば、**example_realm**。
3. **Clients** をクリックしてから、JBoss EAP 用に設定した **client-name** をクリックします。たとえば、**my_jbeap**。
4. **Roles**、**Add Role** の順にクリックします。
5. **example_role** などのロール名を入力し、**Save** をクリックします。これは、JBoss EAP で承認のために設定するロール名です。
6. **Users**、**View all users** の順にクリックします。
7. ID をクリックして、作成したロールを割り当てます。たとえば、**jane_doe** の ID をクリックします。
8. **Role Mappings** をクリックします。**Client Roles** フィールドで、JBoss EAP 用に設定した **client-name** を選択します。たとえば、**my_jbeap**。
9. **Available Roles** で、割り当てるロールを選択します。たとえば、**example_role**。**Add selected** をクリックします。

検証

1. ブラウザーで、アプリケーションの URL に移動します。
2. クレデンシャルを使用してログインします。以下に例を示します。

   ```
   username: jane_doe
   password: janedoep@$$
   ```

   次の出力が得られます。

   ```
   Secured Servlet
   Current Principal '5cb0c4ca-0477-44c3-bdef-04db04d7e39d'
   ```

   必要なロールを持つユーザーは、アプリケーションにログインできます。

関連情報

- Red Hat Single Sign-On でのロールとグループを使用したアクセス許可とアクセスの割り当て

9.4. OPENID CONNECT を使用した JBOSS EAP 起動可能な JAR アプリケーションの開発

OpenID Connect (OIDC) を使用して、認証を外部の OpenID プロバイダーに委任できます。**elytron-oidc-client** galleon レイヤーは、外部の OpenID プロバイダーと接続するための JBoss EAP 起動可能な jar アプリケーションのネイティブ OIDC クライアントを提供します。

Red Hat Single Sign-On を使用して OpenID Connect でセキュリティー保護されたアプリケーションを作成するには、次の手順に従います。

- OpenID プロバイダーとして Red Hat Single Sign-On を設定する
- アプリケーションの Maven プロジェクトを作成する
- OpenID Connect を使用する起動可能な jar アプリケーションを作成する
- ユーザーのロールに基づいてアプリケーションへのアクセスを制限する
- Red Hat Single Sign-On でユーザーロールを作成して割り当てる

9.4.1. OpenID プロバイダーとしての Red Hat Single Sign-On の設定

Red Hat Single Sign-On は、Single Sign-On (SSO) で Web アプリケーションをセキュリティー保護するための ID およびアクセス管理プロバイダーです。OpenID Connect (OAuth 2.0 のエクステンション) をサポートしています。

前提条件

- Red Hat Single Sign-On サーバーをインストールしました。詳細については、Red Hat Single Sign-On **Getting Started Guide** の Installing the Red Hat Single Sign-On server を参照してください。
- Red Hat Single Sign-On サーバーインスタンスにユーザーを作成しました。詳細については、Red Hat Single Sign-On **Getting Started Guide** の Creating a user を参照してください。

手順

1. JBoss EAP のデフォルトポートは 8080 であるため、Red Hat Single Sign-On サーバーを 8080 以外のポートで起動します。

   構文

   ```
   $ RH_SSO_HOME/bin/standalone.sh -Djboss.socket.binding.port-offset=<offset-number>
   ```

   例:

   ```
   $ /home/servers/rh-sso-7.4/bin/standalone.sh -Djboss.socket.binding.port-offset=100
   ```

2. **http://localhost:<port>/auth/** で管理コンソールにログインします。たとえば、**http://localhost:8180/auth/** です。
3. レルムを作成するには、管理コンソールで **Master** にカーソルを合わせ、**Add realm** をクリックします。
4. レルムの名前を入力します。たとえば、**example_realm**。**Enabled** が **オン** になっていることを確認し、**Create** をクリックします。
5. **Users** をクリックし、ユーザーの **Add user** をクリックしてユーザーをレルムに追加します。

6. ユーザー名を入力。たとえば、**jane_doe**。User **Enabled** が **ON** になっていることを確認し、**Save** をクリックします。

7. **Credentials** をクリックして、ユーザーにパスワードを追加します。

8. ユーザーのパスワードを設定します。たとえば、**janedoep@$$**。**Temporary** を **OFF** に切り替えて、**Set Password** をクリックします。確認プロンプトで、**Set password** をクリックします。

9. **Clients** をクリックし、**Create** をクリックしてクライアント接続を設定します。

10. クライアント ID を入力します。たとえば、**my_jbeap**。**Client Protocol** が **openid-connect** に設定されていることを確認し、**Save** をクリックします。

11. **Installation** をクリックし、**Format Option** として **Keycloak OIDC JSON** を選択して、接続パラメーターを確認します。

```
{
  "realm": "example_realm",
  "auth-server-url": "http://localhost:8180/auth/",
  "ssl-required": "external",
  "resource": "my_jbeap",
  "public-client": true,
  "confidential-port": 0
}
```

Red Hat Single Sign-On をアイデンティティプロバイダーとして使用するように JBoss EAP アプリケーションを設定する場合は、次のようにパラメーターを使用します。

```
"provider-url" : "http://localhost:8180/auth/realms/example_realm",
"ssl-required": "external",
"client-id": "my_jbeap",
"public-client": true,
"confidential-port": 0
```

12. **Clients** をクリックし、**my_jbeap** の横にある **Edit** をクリックしてクライアント設定を編集します。

13. **Valid Redirect URIs** に、認証が成功した後にページがリダイレクトする URL を入力します。この例では、この値を **http://localhost:8080/simple-oidc-layer-example/secured/*** に設定します。

関連情報

- セキュアなアプリケーションを作成するための Maven プロジェクトの設定
- レルムおよびユーザーの作成

9.4.2. 起動可能な jar OIDC アプリケーション用の Maven プロジェクトの設定

OpenID Connect を使用する起動可能な jar アプリケーションを作成するために必要な依存関係とディレクトリー構造を使用して Maven プロジェクトを作成します。**elytron-oidc-client** galleon レイヤーは、OpenID プロバイダーと接続するためのネイティブ OpenID Connect (OIDC) クライアントを提供します。

前提条件

- Maven がインストールされている。詳細については、 Downloading Apache Maven を参照してください。
- 最新リリース用に Maven リポジトリーを設定しました。詳細は、 Maven and the JBoss EAP microprofile Maven repository を参照してください。

手順

1. **mvn** コマンドを使用して Maven プロジェクトを設定します。このコマンドは、プロジェクトのディレクトリー構造と **pom.xml** 設定ファイルを作成します。

   構文

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

   例

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

2. アプリケーションのルートディレクトリーに移動します。

   構文

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

   例

   ```
   $ cd simple-oidc-layer-example
   ```

3. 生成された **pom.xml** ファイルを次のように更新します。

   a. 次のリポジトリーを設定します。

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

b. 次のプラグインリポジトリーを設定します。

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

c. 以下のプロパティーを設定します。

```
<properties>
   <maven.compiler.source>1.8</maven.compiler.source>
   <maven.compiler.target>1.8</maven.compiler.target>
   <bootable.jar.maven.plugin.version>6.1.2.Final-redhat-
00001</bootable.jar.maven.plugin.version>
   <jboss.xp.galleon.feature.pack.version>4.0.0.GA-redhat-
00002</jboss.xp.galleon.feature.pack.version>
</properties>
```

d. 次の依存関係を設定します。

```
<dependencies>
   <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0.redhat-1</version>
    <scope>provided</scope>
   </dependency>
</dependencies>

<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.jboss.bom</groupId>
         <artifactId>jboss-eap-jakartaee8</artifactId>
         <version>7.3.4.GA</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
      <dependency>
         <groupId>org.jboss.spec.javax.servlet</groupId>
         <artifactId>jboss-servlet-api_4.0_spec</artifactId>
         <scope>provided</scope>
      </dependency>
   </dependencies>
</dependencyManagement>
```

e. **pom.xml** ファイルの **<build>** 要素で次のビルド設定を設定します。

```
<finalName>${project.artifactId}</finalName>
<plugins>
```

```
  <plugin>
    <groupId>org.wildfly.plugins</groupId>
    <artifactId>wildfly-jar-maven-plugin</artifactId> (1)
    <version>${bootable.jar.maven.plugin.version}</version>
    <configuration>
      <feature-pack-location>org.jboss.eap:wildfly-galleon-
pack:${jboss.xp.galleon.feature.pack.version}</feature-pack-location>
      <layers>
        <layer>jaxrs-server</layer>
        <layer>elytron-oidc-client</layer> (2)
      </layers>
      <context-root>false</context-root> (3)
    </configuration>
    <executions>
      <execution>
        <goals>
          <goal>package</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
```

1. アプリケーションを起動可能な JAR としてビルドするための JBoss EAP Maven プラグイン
2. **elytron-oidc-client** レイヤーは、外部の OpenID プロバイダーと接続するためのネイティブ OpenID Connect (OIDC) クライアントを提供します。
3. **simple-oidc-layer-example** リソースパスにアプリケーションを登録します。サーブレットは、URL **http://server-url/application_name/servlet_path** で利用できます (例: **http://localhost:8080/simple-oidc-layer-example/secured**)。デフォルトでは、アプリケーション WAR ファイルは **http://server-url/servlet_path** のように root-context パスで登録されます (例: **http://localhost:8080/secured**)。

f. **pom.xml** ファイルの **<build>** 要素に simple-oidc-layer-example のようなアプリケーション名を設定します。

```
<finalName>simple-oidc-layer-example</finalName>
```

検証

- アプリケーションのルートディレクトリーで、次のコマンドを入力します。

```
$ mvn install
```

次のような出力が得られます。

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

```
[INFO] Total time: 19.157 s
[INFO] Finished at: 2022-03-10T09:38:21+05:30
[INFO] ------------------------------------------------------------------------
```

これで、OpenID Connect を使用する起動可能な jar アプリケーションを作成できます。

9.4.3. OpenID Connect を使用する起動可能な jar アプリケーションの作成

次の例は、ログインしたユーザーのプリンシパルを出力するサーブレットの作成を示しています。既存のアプリケーションの場合、デプロイメント設定の更新に関連するステップのみが必要です。

この例では、プリンシパルの値は OpenID プロバイダーの ID トークンから取得されます。デフォルトでは、プリンシパルはトークンからの **"sub"** クレームの値です。次のいずれかで、ID トークンからプリンシパルとして使用するクレーム値を指定できます。

- **elytron-oidc-client** サブシステム属性 **principal-attribute**。
- **oidc.json** ファイル。

プロシージャの <application_root> は、**pom.xml** ファイルディレクトリーを示します。**pom.xml** ファイルには、アプリケーションの Maven 設定が含まれています。

前提条件

- Maven プロジェクトを作成しました。詳細は、Configuring Maven project for creating a secure application を参照してください。
- Red Hat Single Sign-On を OpenID プロバイダーとして設定しました。詳細は、Configuring Red Hat Single Sign-On as an OpenID provider を参照してください。

手順

1. Java ファイルを保存するディレクトリーを作成します。

   構文

   ```
   $ mkdir -p <application_root>/src/main/java/com/example/oidc
   ```

   例

   ```
   $ mkdir -p simple-oidc-layer-example/src/main/java/com/example/oidc
   ```

2. 新しいディレクトリーに移動します。

   構文

   ```
   $ cd <application_root>/src/main/java/com/example/oidc
   ```

   例

   ```
   $ cd simple-oidc-layer-example/src/main/java/com/example/oidc
   ```

3. 次の内容のサーブレット SecuredServlet.java を作成します。

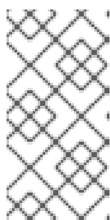
```
package com.example.oidc;

import java.io.IOException;
import java.io.PrintWriter;
import java.security.Principal;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * A simple secured HTTP servlet.
 *
 */
@WebServlet("/secured")
public class SecuredServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        try (PrintWriter writer = resp.getWriter()) {
            writer.println("<html>");
            writer.println("  <head><title>Secured Servlet</title></head>");
            writer.println("  <body>");
            writer.println("    <h1>Secured Servlet</h1>");
            writer.println("    <p>");
            writer.print(" Current Principal '");
            Principal user = req.getUserPrincipal();
            writer.print(user != null ? user.getName() : "NO AUTHENTICATED USER");
            writer.print("'");
            writer.println("    </p>");
            writer.println("  </body>");
            writer.println("</html>");
        }
    }

}
```

4. アプリケーションの **WEB-INF** ディレクトリーにあるデプロイメント記述子 **web.xml** ファイルに、アプリケーションにアクセスするためのセキュリティールールを追加します。

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

<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   metadata-complete="false">

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>secured</web-resource-name>
            <url-pattern>/secured</url-pattern>
```

```
    </web-resource-collection>
    <auth-constraint>
      <role-name>*</role-name>
    </auth-constraint>
  </security-constraint>

  <security-role>
    <role-name>*</role-name>
  </security-role>
</web-app>
```

5. OpenID Connect を使用してアプリケーションをセキュリティー保護するには、デプロイメント設定を更新するか、**elytron-oidc-client** サブシステムを設定します。



注記

デプロイメント設定と **elytron-oidc-client** サブシステムの両方で OpenID Connect を設定する場合、**elytron-oidc-client** サブシステムの **secure-deployment** 属性の設定は、アプリケーションデプロイメント記述子の設定よりも優先されます。

- デプロイメント設定の更新:

  i. 次のように、**WEB-INF** ディレクトリーにファイル **oidc.json** を作成します。

```
{
  "provider-url" : "http://localhost:8180/auth/realms/example_realm",
  "ssl-required": "external",
  "client-id": "my_jbeap",
  "public-client": true,
  "confidential-port": 0
}
```

  ii. デプロイメント記述子 **web.xml** ファイルを次のテキストで更新して、このアプリケーションが OIDC を使用することを宣言します。

```
<login-config>
  <auth-method>OIDC</auth-method>
</login-config>
```

- **elytron-oidc-client** サブシステムの設定:

  i. CLI スクリプトをアプリケーションのルートディレクトリーに保存するディレクトリーを作成します。

構文

```
$ mkdir <application_root>/<cli_script_directory>
```

例

```
$ mkdir simple-oidc-layer-example/scripts/
```

ディレクトリーは、アプリケーションのルートディレクトリー内の Maven がアクセスできる任意の場所に作成できます。

ii. 次の内容で、**configure-oidc.cli** などの CLI スクリプトを作成します。

```
/subsystem=elytron-oidc-client/secure-deployment=simple-oidc-layer-example.war:add(client-id=my_jbeap,provider-url=http://localhost:8180/auth/realms/example_realm,public-client=true,ssl-required=external)
```

サブシステムコマンドは、**simple-oidc-layer-example.war** リソースを、**elytron-oidc-client** サブシステムでセキュリティー保護するためのデプロイメントとして定義します。

iii. プロジェクトの **pom.xml** ファイルで、以下の設定抽出を既存のプラグイン **<configuration>** 要素に追加します。

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

6. アプリケーションのルートディレクトリーで、次のコマンドを使用してアプリケーションをコンパイルします。

```
$ mvn package
```

7. 次のコマンドを使用して、起動可能な jar アプリケーションをデプロイします。

構文

```
$ java -jar <application_root>/target/simple-oidc-layer-example-bootable.jar
```

例

```
$ java -jar simple-oidc-layer-example/target/simple-oidc-layer-example-bootable.jar
```

これにより、JBoss EAP が起動し、アプリケーションがデプロイされます。

検証

1. ブラウザーで、**http://localhost:8080/simple-oidc-layer-example/secured** に移動します。

2. クレデンシャルを使用してログインします。以下に例を示します。

```
username: jane_doe
password: janedoep@$$
```

次の出力が得られます。

```
Secured Servlet
Current Principal '5cb0c4ca-0477-44c3-bdef-04db04d7e39d'
```

これで、Red Hat Single Sign-On で OpenID プロバイダーとして設定したクレデンシャルを使用してアプリケーションにログインできます。

関連情報

- JBoss EAP での OpenID Connect 設定
- ユーザーのロールに基づいてアプリケーションへのアクセスを制限する

9.4.4. 起動可能な jar OIDC アプリケーションのユーザーロールに基づいてアクセスを制限する

ユーザーのロールに基づいて、アプリケーションのすべてまたは一部へのアクセスを制限できます。たとえば、public ロールを持つユーザーに、機密性の低いアプリケーションの部分へのアクセスを許可し、admin ロールを持つユーザーに、機密性の高い部分へのアクセス権を与えることができます。

前提条件

- OpenID Connect を使用してアプリケーションをセキュリティー保護しました。詳細は、Creating a bootable jar application that uses OpenID Connect を参照してください。

手順

1. デプロイメント記述子 **web.xml** ファイルを次のテキストで更新します。

   構文

   ```
   <security-constraint>
       ...
       <auth-constraint>
           <role-name><allowed_role></role-name>
       </auth-constraint>
   </security-constraint>
   ```

   例

   ```
   <security-constraint>
       ...
       <auth-constraint>
           <role-name>example_role</role-name> (1)
       </auth-constraint>
   </security-constraint>
   ```

   (1) ロール **example_role** を持つユーザーのみにアプリケーションへのアクセスを許可します。

2. アプリケーションのルートディレクトリーで、次のコマンドを使用してアプリケーションを再コンパイルします。

```
$ mvn package
```

3. アプリケーションをデプロイします。

```
$ java -jar simple-oidc-layer-example/target/simple-oidc-layer-example-bootable.jar
```

これにより、JBoss EAP が起動し、アプリケーションがデプロイされます。

検証

1. ブラウザーで、**localhost:8080/simple-oidc-layer-example/secured** に移動します。

2. クレデンシャルを使用してログインします。以下に例を示します。

```
username: jane_doe
password: janedoep@$$
```

次の出力が得られます。

```
Forbidden
```

ユーザー jane_doe に必要なロールを割り当てていないため、jane_doe はアプリケーションにログインできません。必要なロールを持つユーザーのみがログインできます。

ユーザーに必要なロールを割り当てるには、Creating and assigning roles to users in Red Hat Single Sign-On を参照してください。

9.4.5. Red Hat Single Sign-On でのユーザーロールの作成と割り当て

Red Hat Single Sign-On は、シングルサインオン (SSO) を使用して Web アプリケーションを保護するためのアイデンティティーおよびアクセス管理プロバイダーです。Red Hat Single Sign-On でユーザーを定義し、ロールを割り当てることができます。

前提条件

- Red Hat Single Sign-On を設定しました。詳細は、Configuring Red Hat Single Sign-On as an OpenID provider を参照してください。

手順

1. **http://localhost:<port>/auth/** で管理コンソールにログインします。たとえば、**http://localhost:8180/auth/** です。

2. JBoss EAP との接続に使用するレルムをクリックします。たとえば、**example_realm**。

3. **Clients** をクリックしてから、JBoss EAP 用に設定した **client-name** をクリックします。たとえば、**my_jbeap**。

4. **Roles**、**Add Role** の順にクリックします。

5. **example_role** などのロール名を入力し、**Save** をクリックします。これは、JBoss EAP で承認のために設定するロール名です。

6. **Users**、**View all users** の順にクリックします。

7. ID をクリックして、作成したロールを割り当てます。たとえば、**jane_doe** の ID をクリックします。
8. **Role Mappings** をクリックします。**Client Roles** フィールドで、JBoss EAP 用に設定した **client-name** を選択します。たとえば、**my_jbeap**。
9. **Available Roles** で、割り当てるロールを選択します。たとえば、**example_role**。**Add selected** をクリックします。

検証

1. ブラウザーで、アプリケーションの URL に移動します。
2. クレデンシャルを使用してログインします。以下に例を示します。

```
username: jane_doe
password: janedoep@$$
```

次の出力が得られます。

```
Secured Servlet
Current Principal '5cb0c4ca-0477-44c3-bdef-04db04d7e39d'
```

必要なロールを持つユーザーは、アプリケーションにログインできます。

関連情報

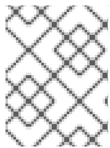
- Red Hat Single Sign-On でのロールとグループを使用したアクセス許可とアクセスの割り当て

第10章 JBOSS EAP での可観測性

開発者またはシステム管理者の場合、**可観測性** は、アプリケーションからの特定の信号に基づいて、アプリケーションの問題の場所と原因を特定するために使用できる一連のプラクティスとテクノロジーです。最も一般的なシグナルは、メトリック、イベント、およびトレースです。JBoss EAP は、**可観測性** のために OpenTelemetry を使用します。

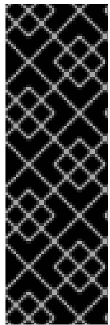
10.1. JBOSS EAP の OPENTELEMETRY

OpenTelemetry は、アプリケーションのテレメトリーデータを計測、生成、収集、およびエクスポートするために使用できるツール、アプリケーションプログラミングインターフェイス (API)、およびソフトウェア開発キット (SDK) のセットです。テレメトリーデータには、メトリック、ログ、およびトレースが含まれます。アプリケーションのテレメトリーデータを分析すると、アプリケーションのパフォーマンスを向上させるのに役立ちます。JBoss EAP は、**opentelemetry** サブシステムを通じて OpenTelemetry 機能を提供します。



注記

Red Hat JBoss Enterprise Application Platform 7.4 は、OpenTelemetry トレース機能のみを提供します。



重要

OpenTelemetry はテクノロジープレビュー機能のみです。テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。Red Hat は、実稼働環境でこれらを使用することを推奨していません。テクノロジープレビューの機能は、最新の技術をいち早く提供して、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。Red Hat のテクノロジープレビュー機能のサポート範囲についての詳細は、https://access.redhat.com/ja/support/offerings/techpreview を参照してください。

関連情報

- OpenTelemetry Documentation

10.2. JBOSS EAP での OPENTELEMETRY 設定

opentelemetry サブシステムを使用して、JBoss EAP で OpenTelemetry のさまざまな側面を設定します。これらには、エクスポーター、スパンプロセッサー、およびサンプラーが含まれます。

exporter

トレースとメトリックを分析および視覚化するには、それらを Jaeger などのコレクターにエクスポートします。Jaeger または OpenTelemetry プロトコル (OTLP) をサポートする任意のコレクターを使用するように JBoss EAP を設定できます。

スパンプロセッサー

スパンプロセッサーは、スパンを生成したとき、またはバッチでエクスポートするように設定できます。エクスポートするトレースの数を設定することもできます。

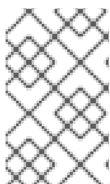
sampler

サンプラーを設定することにより、記録するトレースの数を設定できます。

設定例

次の XML は、デフォルト値を含む完全な OpenTelemetry 設定の例です。変更を加えても JBoss EAP はデフォルト値を保持しないため、設定が異なる場合があります。

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



注記

OpenShift ルートオブジェクトを使用して Jaeger エンドポイントに接続することはできません。代わりに、**http://<ip_address>:<port>** または **http://<service_name>:<port>** を使用してください。

関連情報

- OpenTelemetry サブシステムの属性

10.3. JBOSS EAP での OPENTELEMETRY トレース

JBoss EAP は OpenTelemetry トレースを提供し、ユーザーリクエストがアプリケーションのさまざまな部分を通過する際の進行状況を追跡するのに役立ちます。トレースを分析することで、アプリケーションのパフォーマンスを向上させ、可用性の問題をデバッグできます。

OpenTelemetry トレースは、次のコンポーネントで設定されています。

Trace

要求がアプリケーションで実行する操作のコレクション。

Span

トレース内の単一の操作。要求、エラー、および期間 (RED) メトリックを提供し、スパンコンテキストを含みます。

スパンコンテキスト

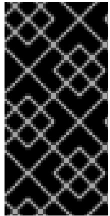
含まれているスパンが一部であるリクエストを表す一意の識別子のセット。

JBoss EAP は、Jakarta RESTful Web Services アプリケーションに対する REST 呼び出し、およびコンテナー管理の Jakarta RESTful Web Services クライアント呼び出しを自動的にトレースします。JBoss EAP は、REST 呼び出しを次のように暗黙的にトレースします。

- 受信要求ごとに:
  - JBoss EAP は、要求からスパンコンテキストをデプロイメントします。
  - JBoss EAP は新しいスパンを開始し、要求が完了するとそれを閉じます。

- 送信要求ごとに:
  - JBoss EAP は、スパンコンテキストを要求に挿入します。
  - JBoss EAP は新しいスパンを開始し、要求が完了するとそれを閉じます。

暗黙的なトレースに加えて、詳細なトレースのために **Tracer** インスタンスをアプリケーションに挿入することにより、カスタムスパンを作成できます。



重要

REST 呼び出し用にエクスポートされた重複トレースが表示される場合は、**microprofile-opentracing-smallrye** サブシステムを無効にします。**microprofile-opentracing-smallrye** を無効にする方法については Removing the microprofile-opentracing-smallrye subsystem を参照してください。

関連情報

- Jaeger を使用してアプリケーションの OpenTelemetry トレースを観察する
- JBoss EAP での OpenTelemetry アプリケーションの開発

10.4. JBOSS EAP で OPENTELEMETRY トレースを有効にする

JBoss EAP で OpenTelemetry トレースを使用するには、最初に **opentelemetry** サブシステムを有効にする必要があります。

前提条件

- JBoss EAP XP がインストールされている。

手順

1. 管理 CLI を使用して OpenTelemetry エクステンションを追加します。

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

2. 管理 CLI を使用して **opentelemetry** サブシステムを有効にします。

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

3. Reload JBoss EAP.

```
reload
```

関連情報

- **opentelemetry** サブシステムの設定

10.5. OPENTELEMETRY サブシステムの設定

opentelemetry サブシステムを設定して、トレースのさまざまな側面を設定できます。トレースの監視に使用するコレクターに基づいてこれらを設定します。

前提条件

- **opentelemetry** サブシステムを有効にしました。詳細は、Enabling OpenTelemetry tracing in JBoss EAP を参照してください。

手順

1. トレースのエクスポータータイプを設定します。

   構文

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

   例

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

2. トレースをエクスポートするエンドポイントを設定します。

   構文

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

   例

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

3. トレースをエクスポートするサービス名を設定します。

   構文

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

   例

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

関連情報

- Jaeger を使用してアプリケーションの OpenTelemetry トレースを観察する

10.6. JAEGER を使用してアプリケーションの OPENTELEMETRY トレースを観察する

JBoss EAP は、Jakarta RESTful Web Services アプリケーションへの REST 呼び出しを自動的かつ暗黙的にトレースします。Jakarta RESTful Web Services アプリケーションに設定を追加したり、**opentelemetry** サブシステムを設定したりする必要はありません。次の手順は、Jaeger コンソールで **helloworld-rs** クイックスタートのトレースを監視する方法を示しています。

前提条件

- Docker をインストールしました。詳細は、Get Docker を参照してください。
- **helloworld-rs** クイックスタートをダウンロードしました。クイックスタートは helloworld-rs で利用可能です。
- **opentelemetry** サブシステムを設定しました。詳細は、Configuring the **opentelemetry** subsystem を参照してください。

手順

1. Docker イメージを使用して Jaeger コンソールを起動します。

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

2. Maven を使用して、ルートディレクトリーから **helloworld-rs** クイックスタートをデプロイします。

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

3. Web ブラウザーで、**http://localhost:8080/helloworld-rs/** のクイックスタートにアクセスし、任意のリンクをクリックします。
4. Web ブラウザーで、**http://localhost:16686/search** にある Jaeger コンソールを開きます。**hello-world.rs** は **Service** の下にリストされています。
5. **hello-world.rs** を選択し、**Find Traces** をクリックします。**hello-world.rs** のトレースの詳細がリスト表示されます。

関連情報

- JBoss EAP での OpenTelemetry アプリケーションの開発

10.7. OPENTELEMETRY トレースアプリケーションの開発

JBoss EAP は Jakarta RESTful Web Services アプリケーションへの REST 呼び出しを自動的かつ暗黙的にトレースしますが、アプリケーションからカスタムスパンを作成して詳細なトレースを行うことができます。**span** は、トレース内の単一の操作です。アプリケーションで、たとえば、リソースが定義されているとき、メソッドが呼び出されているときなどに、スパンを作成できます。**Tracer** インスタンスを挿入することにより、アプリケーションでカスタムトレースを作成します。

10.7.1. OpenTelemetry トレース用の Maven プロジェクトの設定

OpenTelemetry トレースアプリケーションを作成するには、必要な依存関係とディレクトリー構造を使用して Maven プロジェクトを作成します。

前提条件

- Maven がインストールされている。詳細については、Downloading Apache Maven を参照してください。
- 最新リリース用に Maven リポジトリーを設定しました。最新の Maven リポジトリーパッチのインストールは、Maven and the JBoss EAP microprofile maven repository を参照してください。

手順

1. CLI で、**mvn** コマンドを使用して Maven プロジェクトを設定します。このコマンドは、プロジェクトのディレクトリー構造と **pom.xml** 設定ファイルを作成します。

   構文

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

   例

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

2. アプリケーションのルートディレクトリーに移動します。

   構文

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

   例

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

3. 生成された **pom.xml** ファイルを更新します。

   a. 以下のプロパティーを設定します。

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

b. 次の依存関係を設定します。

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

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

    <dependency>
        <groupId>io.opentelemetry</groupId>
        <artifactId>opentelemetry-api</artifactId>
        <version>1.5.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

c. **mvn widlfy:deploy** を使用してアプリケーションをデプロイするには、次のビルド設定を設定します。

```
<build>
  <!-- Set the name of the archive -->
  <finalName>${project.artifactId}</finalName>
  <plugins>
    <!-- Allows to use mvn wildfly:deploy -->
    <plugin>
      <groupId>org.wildfly.plugins</groupId>
      <artifactId>wildfly-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

検証

- アプリケーションのルートディレクトリーで、次のコマンドを入力します。

```
$ mvn install
```

次のような出力が得られます。

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.440 s
[INFO] Finished at: 2021-12-27T14:45:12+05:30
[INFO] ------------------------------------------------------------------------
```

これで、OpenTelemetry トレースアプリケーションを作成できます。

関連情報

- カスタムスパンを作成するアプリケーションの作成

10.7.2. カスタムスパンを作成するアプリケーションの作成

次の手順は、次のような 2 つのカスタムスパンを作成できるアプリケーションを作成する方法を示しています。

- **prepare-hello** - アプリケーションのメソッド **getHello()** が呼び出されたとき。
- **process-hello** - 値 **hello** が新しい **String** オブジェクト **hello** に割り当てられたとき。

この手順では、Jaeger コンソールでこれらのスパンを表示する方法も示します。プロシージャ内の **<application_root>** は、アプリケーションの Maven 設定を含む **pom.xml** ファイルを含むディレクトリーを示します。

前提条件

- Docker をインストールしました。詳細は、Get Docker を参照してください。
- Maven プロジェクトを作成しました。詳細は、Configuring Maven project for OpenTelemetry tracing を参照してください。
- **opentelemetry** サブシステムを設定しました。詳細は、Configuring the **opentelemetry** subsystem を参照してください。

手順

1. **<application_root>** で、Java ファイルを保存するディレクトリーを作成します。

   構文

   ```
   $ mkdir -p src/main/java/com/example/opentelemetry
   ```

   例

   ```
   $ mkdir -p src/main/java/com/example/opentelemetry
   ```

2. 新しいディレクトリーに移動します。

   構文

   ```
   $ cd src/main/java/com/example/opentelemetry
   ```

   例

   ```
   $ cd src/main/java/com/example/opentelemetry
   ```

3. 次の内容の **JakartaRestApplication.java** ファイルを作成します。この **JakartaRestApplication** クラスは、アプリケーションを Jakarta RESTful Web Services アプリケーションとして宣言します。

```
package com.example.opentelemetry;

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

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

4. クラス **ExplicitlyTracedBean** の次のコンテンツを含む **ExplicitlyTracedBean.java** ファイルを作成します。このクラスは、**Tracer** クラスを挿入することによってカスタムスパンを作成します。

```
package com.example.opentelemetry;

import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.Tracer;

@RequestScoped
public class ExplicitlyTracedBean {

    @Inject
    private Tracer tracer; (1)

    public String getHello() {
        Span prepareHelloSpan = tracer.spanBuilder("prepare-hello").startSpan(); (2)
        prepareHelloSpan.makeCurrent();

        String hello = "hello";

        Span processHelloSpan = tracer.spanBuilder("process-hello").startSpan(); (3)
        processHelloSpan.makeCurrent();

        hello = hello.toUpperCase();

        processHelloSpan.end();
        prepareHelloSpan.end();

        return hello;
    }
}
```

1. **Tracer** クラスを挿入して、カスタムスパンを作成します。
2. **getHello()** メソッドが呼び出されたことを示すために、**prepare-hello** というスパンを作成します。
3. **process-hello** というスパンを作成して、値 **hello** が **hello** という新しい **String** オブジェクトに割り当てられたことを示します。

5. **TracedResource** クラスの次のコンテンツを含む **TracedResource.java** ファイルを作成します。このファイルは、**ExplicitlyTracedBean** クラスを挿入し、**traced** と **cdi-trace** の 2 つのエンドポイントを宣言します。

```
package com.example.opentelemetry;

import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
@RequestScoped
public class TracedResource {
    @Inject
    private ExplicitlyTracedBean tracedBean;

    @GET
    @Path("/traced")
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }

    @GET
    @Path("/cdi-trace")
    @Produces(MediaType.TEXT_PLAIN)
    public String cdiHello() {
        return tracedBean.getHello();
    }
}
```

6. アプリケーションのルートディレクトリーに移動します。

構文

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

例

```
$ cd ~/applications/simple-tracing-example
```

7. 次のコマンドを使用して、アプリケーションをコンパイルおよびデプロイします。

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

8. Jaeger コンソールを起動します。

```
$ docker run -d --name jaeger \
  -e COLLECTOR_ZIPKIN_HOST_PORT=:9411 \
  -p 5775:5775/udp \
  -p 6831:6831/udp \
```

```
-p 6832:6832/udp \
-p 5778:5778 \
-p 16686:16686 \
-p 14268:14268 \
-p 14250:14250 \
-p 9411:9411 \
jaegertracing/all-in-one:1.29
```

9. ブラウザーで、**localhost:8080/simple-tracing-example/hello/cdi-trace** に移動します。
10. ブラウザーで、**http://localhost:16686/search** にある Jaeger コンソールを開きます。
11. Jaeger コンソールで、**JBoss EAP XP** を選択し、**Find Traces** をクリックします。
12. **3 Spans** をクリックします。
13. Jaeger コンソールには、次のトレースが表示されます。

```
|GET /hello/cdi-trace (1)
-
 | prepare-hello (2)
  -
  | process-hello (3)
```

(1) これは、自動暗黙的トレースのスパンです。

(2) カスタムスパン **prepare-hello** は、メソッド **getHello()** が呼び出されたことを示します。これは、自動暗黙的トレースのスパンの子です。

(3) カスタムスパン **process-hello** は、値 **hello** が新しい **String** オブジェクト **hello** に割り当てられたことを示します。これは **prepare-hello** スパンの子です。

http://localhost:16686/search でアプリケーションエンドポイントにアクセスするたびに、すべての子スパンで新しいトレースが作成されます。

関連情報

- JBoss EAP での OpenTelemetry トレース

第11章 参照資料

11.1. MICROPROFILE CONFIG リファレンス

11.1.1. デフォルトの MicroProfile Config 属性

MicroProfile Config 仕様はデフォルトで 3 つの **ConfigSource** 定義します。

ConfigSources は、通常の番号に従って並べ替えられます。後のデプロイメントのために設定を上書きする必要がある場合は、ordinal の **ConfigSource** が低いほど、より高い ordinal の **ConfigSource** が上書きされる前に上書きされます。

表11.1 デフォルトの MicroProfile Config 属性

ConfigSource	ordinal
システムプロパティー	**400**
環境変数	**300**
プロパティーファイル **META-INF/microprofile-config.properties** はクラスパスにあります。	**100**

11.1.2. MicroProfile Config SmallRye ConfigSources

microprofile-config-smallrye プロジェクトは、デフォルトの MicroProfile Config **ConfigSource** に加えて使用できる **ConfigSource** を定義します。

表11.2 追加の MicroProfile Config 属性

ConfigSource	ordinal
サブシステムの **config-source**	**100**
ディレクトリーからの **ConfigSource**	**100**
クラスからの **ConfigSource**	**100**

これらの **ConfigSource** には明示的な ordinal が指定されていません。MicroProfile Config 仕様にあるデフォルトの ordinal 値は継承されます。

11.2. MICROPROFILE FAULT TOLERANCE リファレンス

11.2.1. MicroProfile Fault Tolerance 設定プロパティー

SmallRye Fault Tolerance 仕様では、MicroProfile Fault Tolerance 仕様に定義されたプロパティーに加えて、以下のプロパティーを定義します。

表11.3 MicroProfile Fault Tolerance 設定プロパティー

プロパティー	デフォルト値	説明
io.smallrye.faulttolerance.mainThreadPoolSize	**100**	スレッドプールの最大スレッド数
io.smallrye.faulttolerance.mainThreadPoolQueueSize	**-1 （無制限）**	スレッドプールが使用するキューのサイズ。

11.3. MICROPROFILE JWT リファレンス

11.3.1. MicroProfile Config JWT 標準プロパティー

microprofile-jwt-smallrye サブシステムは以下の MicroProfile Config 標準プロパティーをサポートします。

表11.4 MicroProfile Config JWT 標準プロパティー

プロパティー	デフォルト	説明
mp.jwt.verify.publickey	NONE	サポートされている形式のいずれかを使用してエンコードされた公開鍵の文字列表現。**mp.jwt.verify.publickey.location** を設定している場合は設定しないでください。
mp.jwt.verify.publickey.location	NONE	公開鍵の場所は、相対パスまたは URL です。**mp.jwt.verify.publickey** を設定している場合は設定しないでください。
mp.jwt.verify.issuer	NONE	検証している JWT トークンの **iss** 要求の想定される値。

microprofile-config.properties の設定例:

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

11.4. MICROPROFILE OPENAPI リファレンス

11.4.1. MicroProfile OpenAPI 設定プロパティー

JBoss EAP は、標準の MicroProfile OpenAPI 設定プロパティーに加え、以下の追加の MicroProfile OpenAPI プロパティーをサポートします。これらのプロパティーは、アプリケーションスコープおよびグローバルスコープの両方に適用できます。

表11.5 JBoss EAP での MicroProfile OpenAPI プロパティー

プロパティー	デフォルト値	説明
mp.openapi.extensions.enabled	**true**	OpenAPI エンドポイントの登録を有効または無効にします。 **false** に設定すると、OpenAPI ドキュメントの生成を無効にします。config サブシステムを使用するか、**/META-INF/microprofile-config.properties** などの設定ファイルの各アプリケーションに対して、グローバルに値を設定できます。 このプロパティーをパラメーター化することで、実稼働や開発などの異なる環境で **microprofile-openapi-smallrye** を選択的に有効または無効にすることができます。 このプロパティーを使用すると、指定の仮想ホストに関連付けられたアプリケーションが MicroProfile OpenAPI モデルを生成するかを制御できます。
mp.openapi.extensions.path	**/openapi**	このプロパティーを使用して、仮想ホストに関連付けられた複数のアプリケーションの OpenAPI ドキュメントを生成することができます。 同じ仮想ホストに関連付けられた各アプリケーションに、個別の **mp.openapi.extensions.path** を設定します。

プロパティー	デフォルト値	説明
mp.openapi.extensions.servers.relative	**true**	自動生成されるサーバーレコードが絶対的なものであるか OpenAPI エンドポイントの場所と相対的であるかを示します。 root 以外のコンテキストパスが存在するところで OpenAPI ドキュメントの利用者が OpenAPI エンドポイントのホストとの関連した REST サービスへの有効な URL を作成できるようにするサーバーレコードが必要です。 値が **true** の場合は、サーバーレコードが OpenAPI エンドポイントの場所に対して相対的であることを示します。生成されたレコードには、デプロイメントのコンテキストパスが含まれます。 **false** に設定すると、JBoss EAP XP は、デプロイメントにアクセスできるすべてのプロトコル、ホスト、およびポートを含むサーバーレコードを生成します。

11.5. MICROPROFILE REACTIVE MESSAGING リファレンス

11.5.1. 外部メッセージングシステムと統合するための MicroProfile リアクティブメッセージングコネクター

次に、MicroProfile Config 仕様で必要なリアクティブメッセージングプロパティーキー接頭辞のリストを示します。

- **mp.messaging.incoming.[channel-name].[attribute]=[value]**
- **mp.messaging.outgoing.[channel-name].[attribute]=[value]**
- **mp.messaging.connector.[connector-name].[attribute]=[value]**

channel-name は **@Incoming.value()** または **@Outgoing.value()** のいずれかであることに注意してください。明確にするために、コネクターメソッドのペアのこの例を見てみましょう。

```
@Outgoing("to")
public int send() {
   int i = // Randomly generated...
   return i;
}
```

```
@Incoming("from")
public void receive(int i) {
  // Process payload
}
```

この例では、必要なプロパティー接頭辞は次のとおりです。

- **mp.messaging.incoming.from**.これは、**receive()** メソッドを定義します。
- **mp.messaging.outgoing.to**.これは **send()** メソッドを定義します。

これは一例であることを忘れないでください。異なるコネクターは異なるプロパティーを認識するため、指定する接頭辞は、設定するコネクターによって異なります。

11.5.2. リアクティブメッセージングストリームとユーザー初期化コード間のデータ交換の例

以下は、リアクティブメッセージングストリームと、ユーザーが **@Channel** および **Emitter** コンストラクトを介してトリガーしたコードとの間のデータ交換の例です。

```
@Path("/")
@ApplicationScoped
class MyBean {
   @Inject @Channel("my-stream")
   Emitter<String> emitter; (1)

   Publisher<String> dest;

   public MyBean() { (2)
   }

   @Inject
   public MyBean(@Channel("my-stream") Publisher<String> dest) {
      this.dest = subscribeAndAllowMultipleSubscriptions(dest);
   }

   private Publisher subscribeAndAllowMultipleSubscriptions(Publisher delegate) {
   } (3) (4) (5)

   @POST
   public PublisherBuilder<String> publish(@FormParam("value") String value) {
      return emitter.send(value);
   }

   @GET
   public Publisher poll() {
      return dest;
   }

   @PreDestroy
   public void close() { (6)

   }
}
```

インラインの詳細:

1. コンストラクターによって挿入されたパブリッシャーをラップします。
2. Java 仕様のコンテキストと依存関係の挿入 (CDI) を満たすには、この空のコンストラクターが必要です。
3. デリゲートにサブスクライブします。
4. 複数のサブスクリプションを処理できるパブリッシャーでデリゲートをラップします。
5. ラッピングパブリッシャーは、デリゲートからのデータを転送します。
6. リアクティブメッセージングが提供するパブリッシャーからアンサブスクライブします。

この例では、MicroProfile Reactive Messaging が **my-stream** メモリーストリームをリッスンしているため、**Emitter** を介して送信されたメッセージは、この挿入されたパブリッシャーで受信されます。ただし、このデータエクスチェンジを成功させるには、次の条件が満たされている必要があることに注意してください。

1. **Emitter.send()** を呼び出す前に、チャネルにアクティブなサブスクリプションが存在する必要があります。この例では、コンストラクターによって呼び出される **subscribeAndAllowMultipleSubscriptions()** メソッドにより、Bean がユーザーコード呼び出しに使用できるようになるまでにアクティブなサブスクリプションが確実に存在することに注意してください。
2. 挿入された **Publisher** には、**Subscription** を 1 つだけ持つことができます。受信側のパブリッシャーを REST 呼び出しで公開する場合、**poll()** メソッドを呼び出すたびに、**dest** パブリッシャーへの新しいサブスクリプションが生成されます。挿入されたデータを各クライアントにブロードキャストするには、独自のパブリッシャーを実装する必要があります。

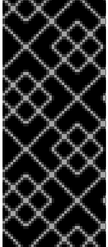
11.5.3. Apache Kafka ユーザー API

Apache Kafka ユーザー API を使用して、Kafka が受信したメッセージに関する詳細情報を取得し、Kafka がメッセージを処理する方法に影響を与えることができます。この API は、io/smallrye/reactive/messaging/kafka/api パッケージに保存されており、次のクラスで設定されています。

- IncomingKafkaRecordMetadata.このメタデータには、次の情報が含まれています。
  - **Message** で表される Kafka レコード **key**。
  - **Message** に使用される Kafka **topic** と **partition**、およびそれら内の **offset**。
  - **Message** の **timestamp** と **timestampType**。
  - **Message headers**。これらは、アプリケーションが生成側で添付し、消費側で受信できる情報です。
- OutgoingKafkaRecordMetadata.このメタデータを使用して、Kafka がメッセージを処理する方法を指定またはオーバーライドできます。次の情報が含まれています。
  - **key**。Kafka はこれをメッセージキーとして扱います。
  - Kafka に使用させたい **topic**。
  - **partition**。

- Kafka が生成する **timestamp** が必要ない場合は、タイムスタンプ。
- **headers**.

- KafkaMetadataUtil には、**OutgoingKafkaRecordMetadata** を **Message** に書き込み、**IncomingKafkaRecordMetadata** を **Message** から読み取るためのユーティリティーメソッドが含まれています。



重要

Kafka にマップされていないチャネルに送信された **Message** に **OutgoingKafkaRecordMetadata** を書き込む場合、リアクティブメッセージングフレームワークはそれを無視します。逆に、Kafka にマップされていないチャネルからの **Message** から **IncomingKafkaRecordMetadata** を読み取ると、そのメッセージは **null** として返されます。

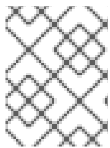
メッセージ key の書き込みと読み取りの方法の例

```
@Inject
@Channel("from-user")
Emitter<Integer> emitter;

@Incoming("from-user")
@Outgoing("to-kafka")
public Message<Integer> send(Message<Integer> msg) {
    // Set the key in the metadata
    OutgoingKafkaRecordMetadata<String> md =
            OutgoingKafkaRecordMetadata.<String>builder()
                .withKey("KEY-" + i)
                .build();
    // Note that Message is immutable so the copy returned by this method
    // call is not the same as the parameter to the method
    return KafkaMetadataUtil.writeOutgoingKafkaMetadata(msg, md);
}

@Incoming("from-kafka")
public CompletionStage<Void> receive(Message<Integer> msg) {
    IncomingKafkaRecordMetadata<String, Integer> metadata =
        KafkaMetadataUtil.readIncomingKafkaMetadata(msg).get();

    // We can now read the Kafka record key
    String key = metadata.getKey();

    // When using the Message wrapper around the payload we need to explicitly ack
    // them
    return msg.ack();
}
```

microprofile-config.properties ファイルの Kafka マッピングの例

```
kafka.bootstrap.servers=kafka:9092

mp.messaging.outgoing.to-kafka.connector=smallrye-kafka
mp.messaging.outgoing.to-kafka.topic=some-topic
mp.messaging.outgoing.to-
```

```
kafka.value.serializer=org.apache.kafka.common.serialization.IntegerSerializer
mp.messaging.outgoing.to-
kafka.key.serializer=org.apache.kafka.common.serialization.StringSerializer

mp.messaging.incoming.from-kafka.connector=smallrye-kafka
mp.messaging.incoming.from-kafka.topic=some-topic
mp.messaging.incoming.from-
kafka.value.deserializer=org.apache.kafka.common.serialization.IntegerDeserializer
mp.messaging.incoming.from-
kafka.key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
```



注記

送信チャネルには **key.serializer** を指定し、受信チャネルには **key.deserializer** を指定する必要があります。

11.5.4. Kafka コネクターの MicroProfile Config プロパティーファイルの例

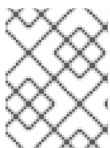
これは、Kafka コネクターのシンプルな **microprofile-config.properties** ファイルの例です。そのプロパティーは、外部メッセージングシステムと統合するための MicroProfile リアクティブメッセージングコネクターの例のプロパティーに対応しています。

```
kafka.bootstrap.servers=kafka:9092

mp.messaging.outgoing.to.connector=smallrye-kafka
mp.messaging.outgoing.to.topic=my-topic
mp.messaging.outgoing.to.value.serializer=org.apache.kafka.common.serialization.IntegerSerializer

mp.messaging.incoming.from.connector=smallrye-kafka
mp.messaging.incoming.from.topic=my-topic
mp.messaging.incoming.from.value.deserializer=org.apache.kafka.common.serialization.IntegerDeseria
zer
```

表11.6 エントリーの議論

エントリー	説明
to、from	これらはチャネルです。
send、receive	これらはメソッドです。 **to** チャネルは **send()** メソッドにあり、**from** チャネルは **receive()** メソッドにあることに注意してください。
kafka.bootstrap.servers=kafka:9092	これは、アプリケーションが接続する必要のある Kafka ブローカーの URL を指定します。次のように、チャネルレベルで URL を指定することもできます: **mp.messaging.outgoing.to.bootstrap.servers=kafka:9092**

エントリー	説明
mp.messaging.outgoing.to.connector=smallrye-kafka	これは **to** チャネルが Kafka からのメッセージを受信することを示しています。 SmallRye リアクティブメッセージングは、アプリケーションをビルドするためのフレームワークです。**smallrye-kafka** 値は、SmallRye リアクティブメッセージング固有であることに注意してください。Galleon を使用して独自のサーバーをプロビジョニングしている場合は、**microprofile-reactive-messaging-kafka** Galleon レイヤーを含めることで、Kafka 統合を有効にできます。
mp.messaging.outgoing.to.topic=my-topic	これは、**my-topic** という Kafka トピックにデータを送信することを示しています。 Kafka のトピックは、メッセージが保存および公開されるカテゴリーまたはフィード名です。すべての Kafka メッセージはトピックに編成されています。プロデューサーアプリケーションはトピックにデータ **to** を書き込み、コンシューマーアプリケーションは **from** トピックからデータを読み取ります。
mp.messaging.outgoing.to.value.serializer=org.apache.kafka.common.serialization.IntegerSerializer	これは、コネクターに **IntegerSerializer** を使用して、**send()** メソッドがトピックに書き込むときに出力する値をシリアル化するように指示します。Kafka は、標準の Java タイプ用のシリアライザーを提供します。**org.apache.kafka.common.serialization.Serializer** を実装するクラスを作成して独自のシリアライザーを実装し、そのクラスをデプロイメントに含めることができます。
mp.messaging.incoming.from.connector=smallrye-kafka	これは、**from** チャネルを使用して Kafka からのメッセージを受信することを示しています。繰り返しますが、**smallrye-kafka** 値は、SmallRye のリアクティブメッセージング固有です。
mp.messaging.incoming.from.topic=my-topic	これは、コネクターが **my-topic** と呼ばれる Kafka トピックからデータを読み取る必要があることを示しています。
mp.messaging.incoming.from.value.deserializer=org.apache.kafka.common.serialization.IntegerDeserializer	これは、**receive()** メソッドを呼び出す前に、**IntegerDeserializer** を使用してトピックから値をデシアライズするようにコネクターに指示します。**org.apache.kafka.common.serialization.Deserializer** を実装するクラスを作成して独自のデシリアライザーを実装し、そのクラスをデプロイメントに含めることができます。



注記

このプロパティーのリストはすべてを網羅したものではありません。詳細は、SmallRye Reactive Messaging Apache Kafka のドキュメントを参照してください。

必須の MicroProfile Reactive Messaging 接頭辞

MicroProfile Reactive Messaging 仕様では、Kafka に次のメソッドプロパティーキー接頭辞が必要です。

- **mp.messaging.incoming.[channel-name].[attribute]=[value]`**
- **mp.messaging.outgoing.[channel-name].[attribute]=[value]`**
- **mp.messaging.connector.[connector-name].[attribute]=[value]`**

channel-name は **@Incoming.value()** または **@Outgoing.value()** のいずれかであることに注意してください。

次に、次のメソッドペアの例を考えてみましょう。

```
@Outgoing("to")
public int send() {
    int i = // Randomly generated...
    return i;
}

@Incoming("from")
public void receive(int i) {
    // Process payload
}
```

このメソッドペアの例では、次の必須のプロパティー接頭辞に注意してください。

- **mp.messaging.incoming.from**.この接頭辞は、**receive()** メソッドの設定としてプロパティーを選択します。
- **mp.messaging.outgoing.to**.この接頭辞は、**send()** メソッドの設定としてプロパティーを選択します。

11.6. OPENID CONNECT リファレンス

11.6.1. elytron-oidc-client サブシステム属性

elytron-oidc-client サブシステムは、その動作を設定するための属性を提供します。

表11.7 elytron-oidc-client サブシステム属性

属性	説明
provider	OpenID Connect プロバイダーの設定。
secure-deployment	OpenID Connect プロバイダーによって保護されたデプロイメント。
realm	Red Hat Single Sign-On レルムの設定。これは便宜上提供されています。keycloak クライアントアダプターで設定をコピーして、ここで使用できます。代わりに **provider** を使用することを推奨します。

重要

現在サポートされていないため、設定で次の **provider**、**realm**、および **secure-deployment** 属性を使用しないでください。

- **autodetect-bearer-only**
- **bearer-only**

現在サポートされていないため、設定で次の **secure-deployment** 属性を使用しないでください。

- **enable-basic-auth**

次の目的で、3 つの **elytron-oidc-client** 属性を使用します。

- **provider**: OpenID Connect プロバイダーを設定します。詳細は、**provider** 属性 を参照してください。
- **secure-deployment**: OpenID Connect によって保護されたデプロイメントを設定します。詳細は、**secure-deployment** 属性 を参照してください。
- **realm**: Red Hat Single Sign-On を設定します。詳細は、 **realm** 属性 を参照してください。**realm** の使用は推奨しません。便宜上提供されています。keycloak クライアントアダプターで設定をコピーして、ここで使用できます。代わりに、**provider** 属性を使用することを推奨します。

表11.8 provider 属性

属性	デフォルト値	説明
allow-any-hostname	**false**	値を **true** に設定すると、OpenID プロバイダーと通信するときにホスト名の検証がスキップされます。これは、テスト時に役立ちます。これを実稼働環境で **ture** に設定しないでください。
always-refresh-token		**true** に設定すると、JBoss EAP はすべての Web 要求でトークンを更新します。
auth-server-url		Red Hat Single Sign-On レルム認証サーバーのベース URL。この属性を使用する場合は、**realm** 属性も定義する必要があります。 または、**provider-url** 属性を使用して、ベース URL とレルムの両方を 1 つの属性で提供することもできます。
client-id		OpenID プロバイダーに登録された JBoss EAP の client-id。
client-key-password		**client-keystore** を指定する場合は、この属性にそのパスワードを指定します。
client-keystore		アプリケーションが HTTPS を介して OpenID プロバイダーと通信する場合は、この属性でクライアントキーストアへのパスを設定します。

属性	デフォルト値	説明
client-keystore-password		**client keystore** を指定する場合は、この属性でそれにアクセスするためのパスワードを指定します。
confidential-port	**8443**	OpenID プロバイダーが使用する機密ポート (SSL/TLS) を指定します。
connection-pool-size		OpenID プロバイダーと通信するときに使用する接続プールのサイズを指定します。
connection-timeout-millis		リモートホストとの接続を確立するためのタイムアウトをミリ秒単位で指定します。最小値は **-1L**、最大値は **2147483647L** です。**-1L** は、値が未定義であることを示します。これがデフォルトです。
connection-ttl-millis		接続を維持する時間をミリ秒単位で指定します。最小値は **-1L**、最大値は **2147483647L** です。**-1L** は、値が未定義であることを示します。これがデフォルトです。
cors-allowed-headers		Cross-Origin Resource Sharing (CORS) が有効になっている場合、これにより **Access-Control-Allow-Headers** ヘッダーの値が設定されます。これはコンマ区切りの文字列である必要があります。これは任意です。設定されていない場合、このヘッダーは CORS 応答で返されません。
cors-allowed-methods		Cross-Origin Resource Sharing (CORS) が有効になっている場合、これにより Access-Control-Allow-Methods ヘッダーの値が設定されます。これはコンマ区切りの文字列である必要があります。これは任意です。設定されていない場合、このヘッダーは CORS 応答で返されません。
cors-exposed-headers		CORS が有効な場合は、Access-Control-Expose-Headers ヘッダーの値を設定します。これはコンマ区切りの文字列である必要があります。これは最適です。設定されていない場合、このヘッダーは CORS 応答で返されません。
cors-max-age		Cross-Origin Resource Sharing (CORS) Max-Age ヘッダーの値を設定します。値は **-1L** から **2147483647L** の間です。この属性は、**enable-cors** が **true** に設定されている場合にのみ有効になります。
disable-trust-manager		HTTPS 経由で OpenID プロバイダーと通信するときにトラストマネージャーを使用するかどうかを指定します。
enable-cors	**false**	Red Hat Single Sign-On Cross-Origin Resource Sharing (CORS) サポートを有効にします。

属性	デフォルト値	説明
expose-token	**false**	**true** に設定すると、認証されたブラウザークライアントは、Javascript HTTP 呼び出し、URL **root/k_query_bearer_token** を介して署名付きアクセストークンを取得できます。これは任意です。これは、Red Hat Single Sign-On に固有のものです。
ignore-oauth-query-parameter	**false**	access_token のクエリーパラメーターの解析を無効にします。
principal-attribute		アイデンティティーのプリンシパルとして使用する ID トークンのクレーム値を指定します。
provider-url		OpenID プロバイダーの URL を指定します。
proxy-url		HTTP プロキシーを使用する場合は、その URL を指定します。
realm-public-key		レルムの公開鍵を指定します。
register-node-at-startup	**false**	**true** に設定すると、登録要求が Red Hat Single Sign-On に送信されます。この属性は、アプリケーションがクラスター化されている場合にのみ役立ちます。
register-node-period		ノードを再登録する頻度を指定します。
socket-timeout-millis		データを待機するソケットのタイムアウトをミリ秒単位で指定します。
ssl-required	**external**	OpenID プロバイダーとの通信を HTTPS 経由で行うかどうかを指定します。値は次のいずれかになります。 ● **all** - すべての通信は HTTPS を介して行われます。 ● **external** - 外部クライアントとの通信のみが HTTP を介して行われます。 ● **none** - HTTP は使用されません。

属性	デフォルト値	説明
token-signature-algorithm	**RS256**	OpenID プロバイダーが使用するトークン署名アルゴリズムを指定します。サポートされているアルゴリズムは次のとおりです。 • **RS256** • **RS384** • **RS512** • **ES256** • **ES384** • **ES512**
token-store		auth-session データの Cookie またはセッションストレージを指定します。
truststore		クライアント HTTPS 要求に使用されるトラストストアを指定します。
truststore-password		トラストストアのパスワードを指定します。
verify-token-audience	**false**	**true** に設定されている場合、bearer-only の認証中に、トークンにこのクライアント名 (**resource**) がオーディエンスとして含まれているかどうかを確認します。

表11.9 secure-deployment 属性

属性	デフォルト値	説明
allow-any-hostname	**false**	値を **true** に設定すると、OpenID プロバイダーと通信するときにホスト名の検証がスキップされます。これは、テスト時に役立ちます。これを実稼働環境で **ture** に設定しないでください。
always-refresh-token		**true** に設定すると、JBoss EAP はすべての Web 要求でトークンを更新します。
auth-server-url		Red Hat Single Sign-On レルム承認サーバーのベース URL。代わりに、**provider-url** 属性を使用することもできます。

属性	デフォルト値	説明
client-id		OpenID プロバイダーに登録された JBoss EAP の client-id。
client-key-password		**client-keystore** を指定する場合は、この属性にそのパスワードを指定します。
client-keystore		アプリケーションが HTTPS を介して OpenID プロバイダーと通信する場合は、この属性でクライアントキーストアへのパスを設定します。
client-keystore-password		**client keystore** を指定する場合は、この属性でそれにアクセスするためのパスワードを指定します。
confidential-port	**8443**	OpenID プロバイダーが使用する機密ポート (SSL/TLS) を指定します。
connection-pool-size		OpenID プロバイダーと通信するときに使用する接続プールのサイズを指定します。
connection-timeout-millis		リモートホストとの接続を確立するためのタイムアウトをミリ秒単位で指定します。最小値は **-1L**、最大値は **2147483647L** です。**-1L** は、値が未定義であることを示します。これがデフォルトです。
connection-ttl-millis		接続を維持する時間をミリ秒単位で指定します。最小値は **-1L**、最大値は **2147483647L** です。-1L は、値が未定義であることを示します。これがデフォルトです。

属性	デフォルト値	説明
cors-allowed-headers		Cross-Origin Resource Sharing (CORS) が有効になっている場合、これにより **Access-Control-Allow-Headers** ヘッダーの値が設定されます。これはコンマ区切りの文字列である必要があります。これは任意です。設定されていない場合、このヘッダーは CORS 応答で返されません。
cors-allowed-methods		Cross-Origin Resource Sharing (CORS) が有効になっている場合、これにより **Access-Control-Allow-Methods** ヘッダーの値が設定されます。これはコンマ区切りの文字列である必要があります。これは任意です。設定されていない場合、このヘッダーは CORS 応答で返されません。
cors-exposed-headers		Cross-Origin Resource Sharing (CORS) が有効になっている場合、これにより **Access-Control-Expose-Headers** ヘッダーの値が設定されます。これはコンマ区切りの文字列である必要があります。これは任意です。設定されていない場合、このヘッダーは CORS 応答で返されません。
cors-max-age		Cross-Origin Resource Sharing (CORS) Max-Age ヘッダーの値を設定します。値は **-1L** から **2147483647L** の間です。この属性は `enable- の場合のみ有効です。
credential		OpenID プロバイダーとの通信に使用する認証情報を指定します。
disable-trust-manager		HTTPS 経由で OpenID プロバイダーと通信するときにトラストマネージャーを使用するかどうかを指定します。

属性	デフォルト値	説明
enable-cors	**false**	Red Hat Single Sign-On Cross-Origin Resource Sharing (CORS) サポートを有効にします。
expose-token	**false**	**true** に設定すると、認証されたブラウザークライアントは、Javascript HTTP 呼び出し、URL **root/k_query_bearer_token** を介して署名付きアクセストークンを取得できます。これはオプションです。これは、Red Hat Single Sign-On に固有です。
ignore-oauth-query-parameter	**false**	access_token のクエリーパラメーターの解析を無効にします。
min-time-between-jwks-requests		アダプターが不明な公開鍵で署名されたトークンを認識すると、JBoss EAP は **elytron-oidc-client** サーバーから新しい公開鍵をダウンロードしようとします。ただし、JBoss EAP は、この属性に設定した値 (秒単位) 未満ですでに新しい公開鍵を試行している場合、新しい公開鍵をダウンロードしようとはしません。値は **-1L** から **2147483647L** の間です。
principal-attribute		アイデンティティーのプリンシパルとして使用する ID トークンのクレーム値を指定します。
provider		OpenID プロバイダーを指定します。
provider-url		OpenID プロバイダーの URL を指定します。
proxy-url		HTTP プロキシーを使用する場合は、その URL を指定します。
public-client	**false**	**true** に設定すると、OpenID プロバイダーとの通信時にクライアント認証情報は送信されません。これは任意です。
realm		Red Hat Single Sign-On で接続するレルム。

属性	デフォルト値	説明
realm-public-key		レルムの公開鍵を指定します。
redirect-rewrite-rule		リダイレクト URI に適用する書き換えルールを指定します。
register-node-at-startup	**false**	**true** に設定すると、登録要求が Red Hat Single Sign-On に送信されます。この属性は、アプリケーションがクラスター化されている場合にのみ役立ちます。
register-node-period		ノードを再登録する頻度を指定します。
resource		OIDC で保護するアプリケーションの名前を指定します。または、**client-id** を指定することもできます。
socket-timeout-millis		データを待機するソケットのタイムアウトをミリ秒単位で指定します。
ssl-required	**external**	OpenID プロバイダーとの通信を HTTPS 経由で行うかどうかを指定します。値は次のいずれかになります。 ● **all** - すべての通信は HTTPS を介して行われます。 ● **external** - 外部クライアントとの通信のみが HTTP を介して行われます。 ● **none** - HTTP は使用されません。
token-minimum-time-to-live		現在のトークンが期限切れになるか、秒単位で設定した時間内に期限切れになる場合、アダプターはトークンを更新します。

属性	デフォルト値	説明
token-signature-algorithm	**RS256**	OpenID プロバイダーが使用するトークン署名アルゴリズムを指定します。サポートされているアルゴリズムは次のとおりです。 • **RS256** • **RS384** • **RS512** • **ES256** • **ES384** • **ES512**
token-store		auth-session データの Cookie またはセッションストレージを指定します。
truststore		アダプタークライアントの HTTPS 要求に使用されるトラストストアを指定します。
truststore-password		トラストストアのパスワードを指定します。
turn-off-change-session-id-on-login	**false**	ログインに成功すると、デフォルトでセッション ID が変更されます。これをオフにするには、値を **true** に設定します。
use-resource-role-mappings	**false**	トークンから取得したリソースレベルの権限を使用します。
verify-token-audience	**false**	**true** に設定すると、bearer-only の認証中に、アダプターはトークンにこのクライアント名 (**resource**) がオーディエンスとして含まれているかどうかを確認します。

表11.10 realm 属性

属性	デフォルト値	説明

属性	デフォルト値	説明
allow-any-hostname	**false**	値を **true** に設定すると、OpenID プロバイダーと通信するときにホスト名の検証がスキップされます。これは、テスト時に役立ちます。これを実稼働環境で **ture** に設定しないでください。
always-refresh-token		**true** に設定すると、JBoss EAP はすべての Web 要求でトークンを更新します。
auth-server-url		Red Hat Single Sign-On レルム承認サーバーのベース URL。代わりに、**provider-url** 属性を使用することもできます。
client-key-password		**client-keystore** を指定する場合は、この属性にそのパスワードを指定します。
client-keystore		アプリケーションが HTTPS を介して OpenID プロバイダーと通信する場合は、この属性でクライアントキーストアへのパスを設定します。
client-keystore-password		**client keystore** を指定する場合は、この属性でそれにアクセスするためのパスワードを指定します。
confidential-port	**8443**	Red Hat Single Sign-On が使用する機密ポート (SSL/TLS) を指定します。
connection-pool-size		Red Hat Single Sign-On と通信するときに使用する接続プールのサイズを指定します。
connection-timeout-millis		リモートホストとの接続を確立するためのタイムアウトをミリ秒単位で指定します。最小値は **-1L**、最大値は **2147483647L** です。**-1L** は、値が未定義であることを示します。これがデフォルトです。

属性	デフォルト値	説明
connection-ttl-millis		接続を維持する時間をミリ秒単位で指定します。最小値は **-1L**、最大値は **2147483647L** です。**-1L** は、値が未定義であることを示します。これがデフォルトです。
cors-allowed-headers		Cross-Origin Resource Sharing (CORS) が有効になっている場合、これにより Access-Control-Allow-Headers ヘッダーの値が設定されます。これはコンマ区切りの文字列である必要があります。これは任意です。設定されていない場合、このヘッダーは CORS 応答で返されません。
cors-allowed-methods		Cross-Origin Resource Sharing (CORS) が有効になっている場合、これにより **Access-Control-Allow-Methods header** の値が設定されます。これはコンマ区切りの文字列である必要があります。これは任意です。設定されていない場合、このヘッダーは CORS 応答で返されません。
cors-exposed-headers		Cross-Origin Resource Sharing (CORS) が有効になっている場合、これにより **Access-Control-Expose-Headers** ヘッダーの値が設定されます。これはコンマ区切りの文字列である必要があります。これは任意です。設定されていない場合、このヘッダーは CORS 応答で返されません。
cors-max-age		Cross-Origin Resource Sharing (CORS) Max-Age ヘッダーの値を設定します。値は **-1L** から **2147483647L** の間です。この属性は、**enable-cors** が **true** に設定されている場合にのみ有効になります。
disable-trust-manager		HTTPS 経由で OpenID プロバイダーと通信するときにトラストマネージャーを使用するかどうかを指定します。

属性	デフォルト値	説明
enable-cors	**false**	{RHProductShortName} Cross-Origin Resource Sharing (CORS) サポートを有効にします。
expose-token	**false**	**true** に設定すると、認証されたブラウザークライアントは、Javascript HTTP 呼び出し、URL **root/k_query_bearer_token** を介して署名付きアクセストークンを取得できます。これは任意です。
ignore-oauth-query-parameter	**false**	access_token のクエリーパラメーターの解析を無効にします。
principal-attribute		アイデンティティーのプリンシパルとして使用する ID トークンのクレーム値を指定します。
provider-url		OpenID プロバイダーの URL を指定します。
proxy-url		HTTP プロキシーを使用する場合は、その URL を指定します。
realm-public-key		レルムの公開鍵を指定します。
register-node-at-startup	**false**	**true** に設定すると、登録要求が Red Hat Single Sign-On に送信されます。この属性は、アプリケーションがクラスター化されている場合にのみ役立ちます。
register-node-period		ノードを再登録する頻度を指定します。
socket-timeout-millis		データを待機するソケットのタイムアウトをミリ秒単位で指定します。

属性	デフォルト値	説明
ssl-required	**external**	OpenID プロバイダーとの通信を HTTPS 経由で行うかどうかを指定します。値は次のいずれかになります。 ● **all** - すべての通信は HTTPS を介して行われます。 ● **external** - 外部クライアントとの通信のみが HTTP を介して行われます。 ● **none** - HTTP は使用されません。
token-signature-algorithm	**RS256**	OpenID プロバイダーが使用するトークン署名アルゴリズムを指定します。サポートされているアルゴリズムは次のとおりです。 ● **RS256** ● **RS384** ● **RS512** ● **ES256** ● **ES384** ● **ES512**
token-store		auth-session データの Cookie またはセッションストレージを指定します。
truststore		クライアント HTTPS 要求に使用されるトラストストアを指定します。
truststore-password		トラストストアのパスワードを指定します。
verify-token-audience	**false**	**true** に設定すると、bearer-only の認証中に、アダプターはトークンにこのクライアント名 (リソース) がオーディエンスとして含まれているかどうかを確認します。

関連情報

- JBoss EAP での OpenID Connect 設定
- Securing applications using OpenID Connect with Red Hat Single Sign-On

11.7. OPENTELEMETRY リファレンス

11.7.1. OpenTelemetry サブシステムの属性

opentelemetry サブシステムの属性を変更して、その動作を設定できます。属性は、設定するアスペクト (exporter、sampler、span processor) ごとにグループ化されています。

表11.11 Exporter 属性グループ

属性	説明	デフォルト値
エンドポイント	OpenTelemetry がトレースをプッシュする URL。これをエクスポーターがリッスンする URL に設定します。	http://localhost:14250/
exporter-type	トレースの送信先のエクスポーター。次のいずれかです。 - **jaeger**.使用するエクスポーターは Jaeger です。 - **otlp**.使用するエクスポーターは、OpenTelemetry プロトコルで動作します。	**jaeger**

表11.12 Sampler 属性グループ

属性	説明	デフォルト値
ratio	トレースとエクスポートの比率。値は **0.0** から **1.0** の間でなければなりません。たとえば、アプリケーションによって作成された 100 トレースごとに 1 つのトレースをエクスポートするには、値を **0.01** に設定します。この属性は、属性 **sampler-type** を **ratio** として設定した場合にのみ有効になります。	

表11.13 Span processor 属性グループ

属性	説明	デフォルト値
batch-delay	JBoss EAP による 2 つの連続したエクスポート間のミリ秒単位の間隔。この属性は、属性 **span-processor-type** を **batch** として設定した場合にのみ有効になります。	**5000**
export-timeout	キャンセルされる前にエクスポートが完了するまでの最大時間 (ミリ秒単位)。	**30000**
max-export-batch-size	各バッチで公開されるトレースの最大数。この数は、**max-queue-size** の値以下である必要があります。この属性を設定できるのは、属性 **span-processor-type** を **batch** として設定した場合のみです。	**512**
max-queue-size	エクスポートする前にキューに入れるトレースの最大数。アプリケーションがさらにトレースを作成する場合、それらは記録されません。この属性は、属性 **span-processor-type** を **batch** として設定した場合にのみ有効になります。	**2048**
span-processor-type	使用する span processor のタイプ。値は次のいずれかになります。 ● **batch**: JBoss EAP は、以下の属性を使用して定義されたバッチでトレースをエクスポートします。 ○ **batch-delay** ○ **max-export-batch-size** ○ **max-queue-size** ● **simple**: JBoss EAP エクスポートトレースは終了するとすぐに実行されます。	**batch**

関連情報

- JBoss EAP の OpenTelemetry