Red Hat JBoss Enterprise Application Platform 7.3

JBoss EAP XP 1.0.0 での Eclipse MicroProfile の使用

JBoss EAP XP 1.0.0 向け

Last Updated: 2023-08-23

Red Hat JBoss Enterprise Application Platform 7.3 JBoss EAP XP 1.0.0 での Eclipse MicroProfile の使用

JBoss EAP XP 1.0.0 向け

法律上の通知

Copyright © 2023 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 での Eclipse MicroProfile の使用に関する一般的な情報を提供します。

目次

第1章 最新の MICROPROFILE 機能向けの JBOSS EAP XP ... 4
1.1. JBOSS EAP XP について 4
1.2. JBOSS EAP XP のインストール 4
1.3. JBOSS EAP XP パッチストリームを管理するための JBOSS EAP XP マネージャー 4
1.4. JBOSS EAP 7.3.0 への JBOSS EAP XP 1.0.0 のインストール 5
1.5. JBOSS EAP 7.3.1 への JBOSS EAP XP 1.0.0 のインストール 5
1.6. JBOSS EAP のアンインストール 6
1.7. JBOSS EAP XP の状態の表示 6

第2章 ECLIPSE MICROPROFILE について ... 8
2.1. ECLIPSE MICROPROFILE CONFIG 8
2.2. ECLIPSE MICROPROFILE FAULT TOLERANCE 9
2.3. ECLIPSE MICROPROFILE HEALTH 10
2.4. ECLIPSE MICROPROFILE 11
2.5. ECLIPSE MICROPROFILE METRICS 12
2.6. ECLIPSE MICROPROFILE OPENAPI 13
2.7. ECLIPSE MICROPROFILE OPENTRACING 13
2.8. ECLIPSE MICROPROFILE REST クライアント 15

第3章 JBOSS EAP での ECLIPSE MICROPROFILE の管理 ... 16
3.1. ECLIPSE MICROPROFILE OPENTRACING 管理 16
3.2. ECLIPSE MICROPROFILE CONFIG 設定 17
3.3. ECLIPSE MICROPROFILE FAULT TOLERANCE 設定 19
3.4. ECLIPSE MICROPROFILE HEALTH 設定 20
3.5. ECLIPSE MICROPROFILE JWT 設定 22
3.6. ECLIPSE MICROPROFILE METRICS 管理 22
3.7. ECLIPSE MICROPROFILE OPENAPI 管理 24
3.8. スタンドアロンサーバー設定 27

第4章 JBOSS EAP の ECLIPSE MICROPROFILE アプリケーションの開発 ... 30
4.1. MAVEN および JBOSS EAP ECLIPSE MICROPROFILE MAVEN リポジトリー 30
4.2. ECLIPSE MICROPROFILE CONFIG の開発 33
4.3. ECLIPSE MICROPROFILE FAULT TOLERANCE アプリケーションの開発 36
4.4. ECLIPSE MICROPROFILE HEALTH の開発 41
4.5. ECLIPSE MICROPROFILE JWT アプリケーション開発 43
4.6. ECLIPSE MICROPROFILE METRICS の開発 49
4.7. ECLIPSE MICROPROFILE OPENAPI アプリケーションの開発 51
4.8. ECLIPSE MICROPROFILE REST クライアントの開発 58

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 ... 63
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 63
5.2. RED HAT コンテナーレジストリーへの認証の設定 64
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 64
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 66
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 68

第6章 RED HAT CODEREADY STUDIO での JBOSS EAP の ECLIPSE MICROPROFILE アプリケーションの開発の有効化 ... 70
6.1. CODEREADY STUDIO での JBOSS EAP XP のインストール 70
6.2. ECLIPSE MICROPROFILE 機能を使用するための CODEREADY STUDIO の設定 71
6.3. CODEREADY STUDIO での ECLIPSE MICROPROFILE クイックスタートの使用 72

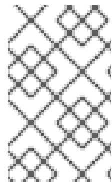
第7章 REFERENCE ... 74

7.1. ECLIPSE MICROPROFILE CONFIG リファレンス 74
7.2. ECLIPSE MICROPROFILE FAULT TOLERANCE リファレンス 74
7.3. ECLIPSE MICROPROFILE JWT リファレンス 75
7.4. ECLIPSE MICROPROFILE OPENAPI リファレンス 75

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 (Eclipse 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 3.3 コンポーネントを提供します。

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

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。最新の JBoss EAP XP 1.0.x パッチは、最新の JBoss EAP 7.3 パッチと互換性があります。

関連情報

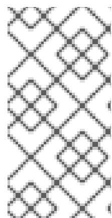
- JBoss EAP 7.3.0 への JBoss EAP XP 1.0.0 のインストール
- JBoss EAP 7.3.1 への JBoss EAP XP 1.0.0 のインストール

1.3. JBOSS EAP XP パッチストリームを管理するための 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 3.3 実装と、これらの MicroProfile 3.3 実装のバグ修正が含まれます。

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一覧と、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 7.3.0 への JBOSS EAP XP 1.0.0 のインストール

JBoss EAP XP 1.0.0 は JBoss EAP 7.3.1 で認定されています。

JBoss EAP XP 1.0.0 を JBoss EAP 7.3.0 サーバーにインストールする場合は、パッチを適用して JBoss EAP 7.3.1 にアップグレードする必要があります。

要件

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-1.0.0-manager.jar** ファイル (JBoss EAP XP マネージャー)
- JBoss EAP 7.3.1 GA パッチ
- JBoss EAP XP 1.0.0 パッチ

手順

1. 以下の管理コマンドを使用して、JBoss EAP 7.3.1 GA パッチを適用します。

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

2. 以下のコマンドを使用して JBoss EAP XP マネージャーを設定します。

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

3. 以下の管理コマンドを使用して、JBoss EAP XP 1.0.0 パッチを適用します。

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

4. サーバーを再起動します。

   ```
   shutdown --restart
   ```

1.5. JBOSS EAP 7.3.1 への JBOSS EAP XP 1.0.0 のインストール

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-1.0.0-manager.jar** ファイル (JBoss EAP XP マネージャー)
- JBoss EAP XP 1.0.0 パッチ

手順

1. 以下のコマンドを使用して JBoss EAP XP マネージャーを設定します。

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

2. 以下の管理コマンドを使用して、JBoss EAP XP 1.0.0 パッチを適用します。

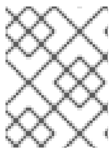
```
patch apply /path/to/jboss-eap-xp-1.0.0-patch.zip
```

3. サーバーを再起動します。

```
shutdown --restart
```

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 1.0.0 パッチストリームと Eclipse MicroProfile 3.3 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 1.0.0 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
```

Eclipse MicroProfile 3.3 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 Eclipse MicroProfile 3.3 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
- 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
- 現在の状態によるサポートポリシーの変更

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
- JBoss EAP 7.3.1 への JBoss EAP XP 1.0.0 のインストール

第2章 ECLIPSE MICROPROFILE について

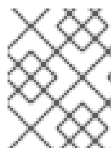
2.1. ECLIPSE MICROPROFILE CONFIG

2.1.1. JBoss EAP の Eclipse 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。

Eclipse 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。

Eclipse 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デフォルトの JBoss EAP 7.3 設定に含まれています。



注記

Eclipse 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。

その他のリソース

- Eclipse MicroProfile Config
- SmallRye Config

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

Eclipse 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。

Eclipse 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.eclipse.microprofile.config.spi.ConfigSource**

2.2. ECLIPSE MICROPROFILE FAULT TOLERANCE

2.2.1. Eclipse MicroProfile Fault Tolerance 仕様について

Eclipse 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。

Eclipse 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。

その他のリソース

- Eclipse MicroProfile Fault Tolerance 仕様

2.2.2. JBoss EAP での Eclipse MicroProfile Fault Tolerance

microprofile-fault-tolerance-smallrye サブシステムは、JBoss EAP での Eclipse 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。
- 呼び出し時では、呼び出しが CDI 仕様に定義された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、Eclipse 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.globalThreadPoolSize**
- **io.smallrye.faulttolerance.timeoutExecutorThreads**

その他のリソース

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

2.3. ECLIPSE MICROPROFILE HEALTH

2.3.1. JBoss EAP の Eclipse 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。

Eclipse 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。

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

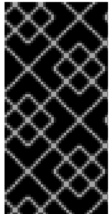
Readiness

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。

以前のバージョンの Eclipse MicroProfile Health 仕様で定義された **@Health** アノテーションが非推奨になりました。



重要

デフォルトでは、**microprofile-health-smallrye** サブシステムはサーバーが稼働していることのみを評価します。**:empty-readiness-checks-status** および **:empty-liveness-checks-status** は、**readiness** または **liveness** 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する管理属性です。

その他のリソース

- 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
- Eclipse MicroProfile Health
- カスタムヘルスチェックの実装

2.4. ECLIPSE MICROPROFILE

2.4.1. JBoss EAP での Eclipse MicroProfile JWT 統合

サブシステム **microprofile-jwt-smallrye** は JBoss EAP で Eclipse 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。

- Eclipse MicroProfile JWT セキュリティーを使用するデプロイメントの検出。
- Eclipse 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 Eclipse MicroProfile JWT 統合を提供します。

その他のリソース

- SmallRye JWT

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

Eclipse 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、Eclipse MicroProfile JWT デプロイメント全体で使用されます。

Eclipse MicroProfile JWT デプロイメントは Eclipse 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 Eclipse MicroProfile JWT アクティベーション

Eclipse 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。

Eclipse 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、Eclipse 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 Eclipse MicroProfile JWT の制限

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

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

- Eclipse 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。

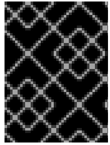
いずれの場合も、**mp.jwt.verify.publickey.location** 設定プロパティーを使用する代わりに、クリアーテキスト JWKS を参照できます。

2.5. ECLIPSE MICROPROFILE METRICS

2.5.1. JBoss EAP の Eclipse 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 Eclipse 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

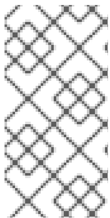
- SmallRye Metrics
- Eclipse MicroProfile Metrics

2.6. ECLIPSE MICROPROFILE OPENAPI

2.6.1. JBoss EAP での Eclipse MicroProfile OpenAPI

Eclipse 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。

Eclipse 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. ECLIPSE MICROPROFILE OPENTRACING

2.7.1. Eclipse 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。

Eclipse 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。

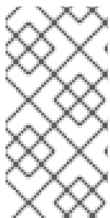
その他のリソース

- Eclipse MicroProfile OpenTracing 仕様の詳細は、Eclipse MicroProfile OpenTracing 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** javadoc を参照してください。

2.7.2. EAP での Eclipse MicroProfile OpenTracing

microprofile-opentracing-smallrye サブシステムを使用して、Jakarta EE アプリケーションを追跡する環境変数を指定できます。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 Eclipse MicroProfile OpenTracing 機能を提供します。

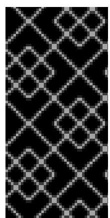
MicroProfile 1.3.0 は、アプリケーションのリクエストのトレースをサポートします。デフォルトの Jaeger Java Client トレーサーや、Jakarta EE で 一 般的に使用されるコンポーネントのインストルメンテーションライブラリーのセットを設定して、システムプロパティーまたは環境変数を設定できます。



注記

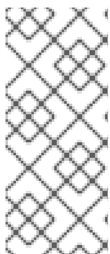
JBoss EAP サーバーに自動的にデプロイされた各 WAR は、独自の **Tracer** インスタンスを持ちます。EAR 内の各 WAR は個別の WAR として扱われ、各 WAR には独自の **Tracer** インスタンスがあります。デフォルトでは、Jaeger Client と使用されるサービス名はデプロイメントの名前から派生し、通常これは WAR ファイル名になります。

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 Jeager 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。Jeager 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。



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** javadoc を参照してください。
- デフォルトトレーサーをオーバーライドする方法と、CDI bean のトレース方法に関する詳細は、**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. ECLIPSE MICROPROFILE REST クライアント

2.8.1. MicroProfile REST クライアント

JBoss EAP XP 1.0.0 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利用するため、JAX-RS 2.1 クライアント上に構築される MicroProfile REST クライアント 1.4.x をサポートするようになりました。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
- サーバー上のヘッダーの伝搬
- **ResponseExceptionMapper**
- CDI の統合

その他のリソース

- MicroProfile REST クライアントと JAX-RS 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 クライアントでのサーバーでヘッダーの伝搬
- 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

第3章 JBOSS EAP での ECLIPSE MICROPROFILE の管理

3.1. ECLIPSE MICROPROFILE OPENTRACING 管理

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 7.3 設定に含まれています。このサブシステムは、JBoss EAP 7.3 の Eclipse 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. 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. microprofile-opentracing-smallrye サブシステムの追加

サーバー設定に追加することで、**microprofile-opentracing-smallrye** サブシステムを有効化できます。管理 CLI で **add** 操作を使用して、指定のサーバーで MicroProfile OpenTracing 機能をグローバルに有効にします。

手順

1. サブシステムを追加します。

```
/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
```

3.1.4. 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. ECLIPSE MICROPROFILE CONFIG 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. ECLIPSE 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. ECLIPSE 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. Eclipse 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. 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 および **:empty-liveness-checks-status** は、**readiness** または **liveness** 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する管理属性です。

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 または live 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}"
}
```

readiness および **liveness** プローブの両方を確認する **/health** HTTP エンドポイントと **:check** 操作は、これらの属性も考慮します。

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. ECLIPSE MICROPROFILE JWT 設定

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

Eclipse 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. ECLIPSE 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クスは Prometheus 形式で公開されます。

メトリクスは 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クスが含まれます。
- **/metrics/vendor**: メモリープールなどのベンダー固有のメトリクスが含まれます。

- **/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クスが含まれます。

メトリクス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クス属性 **request-count** を公開します。このメトリクスの名前は **jboss_undertow_request_count** です。接頭辞 **jboss** は JBoss EAP をメトリクス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クスの検証

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クスを確認します。

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. Eclipse 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クスを公開する 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
```

- 既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. ECLIPSE MICROPROFILE OPENAPI 管理

3.7.1. Eclipse 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 Eclipse 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 Eclipse 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 Eclipse 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。

要件

- クエリーされるデプロイメントは、Eclipse 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 Eclipse 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 Eclipse 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。

要件

- クエリーされるデプロイメントは、Eclipse 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 JAX-RS 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. スタンドアロンサーバー設定

3.8.1. 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 1.0.0 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、Eclipse 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。 • EJB • Messaging • Java Batch • JavaServer Faces • EJB 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、Eclipse 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 Eclipse 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

3.8.2. Eclipse 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 Eclipse 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 Eclipse 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-openapi-smallrye**
- **microprofile-jwt-smallrye**
- **microprofile-fault-tolerance-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 Eclipse 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 ECLIPSE MICROPROFILE アプリケーションの開発

4.1. MAVEN および JBOSS EAP ECLIPSE 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 Eclipse MicroProfile Maven リポジトリーパッチのダウンロード

Eclipse MicroProfile Expansion Pack が JBoss EAP に対してリリースされるたびに、JBoss EAP Eclipse 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 7.3.0 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. 一覧で **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 **1.0.0** を選択します。
6. **Release** タブをクリックします。
7. 一覧で **JBoss EAP XP 1.0.0 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 Eclipse 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 Eclipse 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。

要件

- Red Hat JBoss Enterprise Application Platform 7.3.0 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 7.3 Maven リポジトリーがインストールされていることを確認します。
- ローカルシステムに JBoss EAP XP 1.0.0 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 7.3.0.GA Maven リポジトリーへのパスを見つけます。例: **/path/to/repo/jboss-eap-7.3.0.GA-maven-repository/maven-repository/**
2. ダウンロードした JBoss EAP XP 1.0.0 Incremental Maven リポジトリーを直接 Red Hat JBoss Enterprise Application Platform 7.3.0.GA Maven リポジトリーのディレクトリーに展開します。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 7.3.0.GA Maven リポジトリーパスの値を置き換えます。

```
$ unzip -o jboss-eap-xp-1.0.0-incremental-maven-repository.zip -d
EAP_MAVEN_REPOSITORY_PATH
```

注記

EAP_MAVEN_REPOSITORY_PATH は **jboss-eap-7.3.0.GA-maven-repository** を参照します。たとえば、この手順は、**/path/to/repo/jboss-eap-7.3.0.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 7.3.0.GA Maven リポジトリーに抽出した後、リポジトリー名は JBoss EAP Eclipse MicroProfile Maven リポジトリーになります。

その他のリソース

- 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 Eclipse MicroProfile BOM

JBoss EAP XP 1.0.0 には JBoss EAP Eclipse MicroProfile BOM が含まれています。この BOM は **jboss-eap-xp-microprofile** という名前で、ユースケースでは JBoss EAP Eclipse MicroProfile API に対応しています。

表4.1 JBoss EAP Eclipse MicroProfile BOM

BOM アーティファクト ID	ユースケース

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 Eclipse 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 Eclipse MicroProfile Maven リポジトリーの使用

Red Hat JBoss Enterprise Application Platform 7.3.0.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 Eclipse 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 Eclipse 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 Eclipse 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。

要件

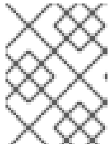
- ローカルシステムに Red Hat JBoss Enterprise Application Platform 7.3 Maven リポジトリーをインストールし、JBoss EAP XP Incremental Maven リポジトリーを適用している。

手順

1. 設定方法を選択し、JBoss EAP Eclipse MicroProfile Maven リポジトリーを設定します。
2. JBoss EAP Eclipse 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>1.0.0.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. ECLIPSE MICROPROFILE CONFIG の開発

4.2.1. Eclipse 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、Eclipse 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。

要件

- 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 Eclipse MicroProfile Config アーティファクトおよび Eclipse 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>1.0.0.GA</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

3. BOM によって管理される Eclipse MicroProfile Config アーティファクトおよび Eclipse 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、Eclipse MicroProfile Config および Eclipse 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。

要件

- JBoss EAP では Eclipse 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、Eclipse 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 JAX-RS 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. ECLIPSE 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. Eclipse 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、Eclipse 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。

要件

- 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 Eclipse 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 Eclipse 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、Eclipse 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. CDI Bean を以下の内容で **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. ECLIPSE 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.Health** アノテーションを含む CDI bean は、実行時に自動的に検出および呼び出しされます。

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

```
@Health
public class HealthTest implements HealthCheck {

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

デプロイされると、以下の例のように、後続のヘルスチェッククエリーにカスタムチェックが含まれます。

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

その他のリソース

- https://openliberty.io/javadocs/microprofile-1.2-javadoc/org/eclipse/microprofile/health/Health.html

4.4.2. @Liveness アノテーションの例

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

```
@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

以下の例は、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");
        }
    }
}
```

4.5. ECLIPSE MICROPROFILE JWT アプリケーション開発

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

Eclipse 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 Eclipse 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。

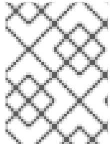
3. BOM によって管理される Eclipse MicroProfile JWT アーティファクトをプロジェクト POM ファイルの **<dependency>** セクションに追加します。以下の例は、Eclipse MicroProfile JWT 依存関係をファイルに追加する方法を示しています。

```
<!-- 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. Eclipse 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、例としてトークンを生成するコードを提供します。独自のトークンジェネレーターを実装する必要があります。

要件

- 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。実稼働環境の場合は、独自のトークンジェネレーターを実装します。

   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 JAX-RS 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 JAX-RS 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. ECLIPSE MICROPROFILE METRICS の開発

4.6.1. Eclipse 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. メトリクスをテストします。

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

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

Maven プロジェクトを作成し、Eclipse 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。

要件

- 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。
  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 てください。

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>1.0.0.GA</version.server.bom>
  </properties>

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

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

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

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。

4.7.3. Eclipse MicroProfile OpenAPI アプリケーションの作成

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。

要件

- Maven プロジェクトは、Eclipse 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一覧は、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 JAX-RS 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. ECLIPSE MICROPROFILE REST クライアントの開発

4.8.1. MicroProfile REST クライアントと JAX-RS 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、JAX-RS をネイティブで **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}")mes("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、**@HeaderValue** 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 クライアントでのサーバーでヘッダーの伝搬

org.eclipse.microprofile.rest.client.ext.ClientHeadersFactory のインスタンスが有効であれば、受信ヘッダーの送信要求に一括転送できます。デフォルトのインスタンス **org.eclipse.microprofile.rest.client.ext.DefaultClientHeadersFactoryImpl** は、コンマ区切りの設定プロパティー **org.eclipse.microprofile.rest.client.propagateHeaders** に一覧表示される着信ヘッダーで設定されるマップを返します。

ClientHeadersFactory インターフェイスをインスタンス化するルールは次のとおりです。

- JAX-RS リクエストのコンテキストで呼び出される **ClientHeadersFactory** インスタンスは、**@Context** アノテーションが付けられたフィールドおよびメソッドの挿入をサポートできます。
- CDI によって管理される **ClientHeadersFactory** インスタンス は、適切な CDI 管理インスタンスを使用する必要があります。**@Inject** インジェクションもサポートする必要があります。

org.eclipse.microprofile.rest.client.ext.ClientHeadersFactory インターフェイスは以下のように定義されます。

```
public interface ClientHeadersFactory {
```

```
/**
 * Updates the HTTP headers to send to the remote service. Note that providers
 * on the outbound processing chain could further update the headers.
 *
 * @param incomingHeaders - the map of headers from the inbound JAX-RS request. This will
 * be an empty map if the associated client interface is not part of a JAX-RS request.
 * @param clientOutgoingHeaders - the read-only map of header parameters specified on the
 * client interface.
 * @return a map of HTTP headers to merge with the clientOutgoingHeaders to be sent to
 * the remote service.
 */
MultivaluedMap<String, String> update(MultivaluedMap<String, String> incomingHeaders,
                                      MultivaluedMap<String, String> clientOutgoingHeaders);
}
```

その他のリソース

- ClientHeadersFactory Javadoc

4.8.6. MicroProfile REST クライアントの ResponseExceptionMapper

org.eclipse.microprofile.rest.client.ext.ResponseExceptionMapper は、JAX-RS で定義される **javax.ws.rs.ext.ExceptionMapper** クラスと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。

4.8.7. 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 CDI 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。例を以下に示します。

```
@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。以下のシステム変数を使用して情報を外部で提供することもできます。

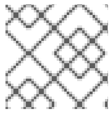
```
<fully qualified name of TestDataBase>/mp-rest/url=<URL>
```

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
```

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

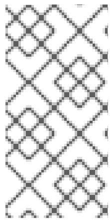
JBoss EAP XP の OpenShift イメージでマイクロサービスアプリケーションをビルドし、実行できます。



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 1.0.0 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 1.0.0 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。

その他のリソース

- 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のインポート

手順

1. 以下のコマンドをいずれか 1 つ使用して、JBoss EAP XP の OpenShift イメージの最新 JDK 8 および JDK 11 イメージストリームとテンプレートを OpenShift プロジェクトの名前空間にインポートします。
   a. JDK 8 イメージストリーム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-xp1/jboss-eap-xp1-openjdk8-openshift.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 8 ビルダーイメージストリーム: jboss-eap-xp1-openjdk8-openshift
- JDK 8 ランタイムイメージストリーム: jboss-eap-xp1-openjdk8-runtime-openshift

b. JDK 11 イメージストリームをインポートします。

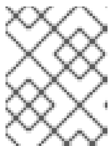
```
oc replace --force -f https://raw.githubusercontent.com/jboss-container-images/jboss-eap-openshift-templates/eap-xp1/jboss-eap-xp1-openjdk11-openshift.json
```

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-xp1-openjdk11-openshift
- JDK 11 ランタイムイメージストリーム: jboss-eap-xp1-openjdk11-runtime-openshift

c. JDK 8 および JDK 11 テンプレートをインポートします。

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



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。例を以下に示します。

```
...
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。例を以下に示します。

```
...
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詳細は、https://docs.openshift.com/container-platform/latest/openshift_images/configuring-

samples-operator.html 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-xp1-basic-s2i \ (1)
 -p EAP_IMAGE_NAME=jboss-eap-xp1-openjdk8-openshift:1.0 \
 -p EAP_RUNTIME_IMAGE_NAME=jboss-eap-xp1-openjdk8-runtime-openshift:1.0 \
 -p IMAGE_STREAM_NAMESPACE=eap-demo \ (2)
 -p SOURCE_REPOSITORY_URL=https://github.com/jboss-developer/jboss-eap-quickstarts
\ (3)
 -p SOURCE_REPOSITORY_REF=xp-1.0.x \ (4)
 -p CONTEXT_DIR=microprofile-config (5)
```

(1) 使用するテンプレート。

(2) 最新のイメージとテンプレートは、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。

別の例として、JDK 11 ランタイムイメージを使用して **microprofile-config** クイックスタートをデプロイするには、以下のコマンドを入力します。このコマンドは、GitHub の **microprofile-config** ソースコードとともに アプリケーションのデプロイメントに向けた OpenShift の準備 セクションで作成した、**eap-demo** プロジェクトで **eap-xp1-basic-s2i** テンプレートを使用します。

```
$ oc new-app --template=eap-xp1-basic-s2i \ (1)
 -p EAP_IMAGE_NAME=jboss-eap-xp1-openjdk11-openshift:1.0 \
 -p EAP_RUNTIME_IMAGE_NAME=jboss-eap-xp1-openjdk11-runtime-openshift:1.0 \
 -p IMAGE_STREAM_NAMESPACE=eap-demo \ (2)
 -p SOURCE_REPOSITORY_URL=https://github.com/jboss-developer/jboss-eap-quickstarts
```

```
\ ③
-p SOURCE_REPOSITORY_REF=xp-1.0.x \ ④
-p CONTEXT_DIR=microprofile-config ⑤
```

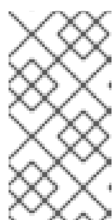
① 使用するテンプレート。

② 最新のイメージとテンプレートは、プロジェクトの名前空間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-xp1-basic-app-build-artifacts

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

その他のリソース

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

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。

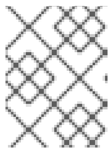
- アプリケーションを OpenShift の外部から表示できるようにサービスを公開します。
- アプリケーションを特定のレプリカ数にスケーリングします。

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

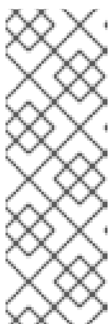
テンプレートを使用してアプリケーションを作成した場合は、ルートがすでに存在することがあります。存在する場合は次のステップに進みます。

```
$ oc expose service/eap-xp1-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 1.0.0 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、JBoss EAP XP 1.0.1 GA ディストリビューションでのみ利用できます。

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-xp1-basic-app --replicas=3
```

その他のリソース

JBoss EAP XP クイックスタートの詳細は、JBoss EAP**JBoss EAP での Eclipse MicroProfile の使用**の クイックスタートの使用 を参照してください。

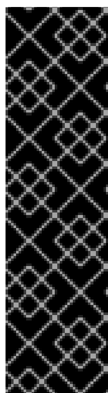
第6章 RED HAT CODEREADY STUDIO での JBOSS EAP の ECLIPSE MICROPROFILE アプリケーションの開発の有効化

CodeReady Studio で開発するアプリケーションに Eclipse MicroProfile 機能を組み込む場合は、CodeReady Studio で JBoss EAP の Eclipse 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 Eclipse 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。



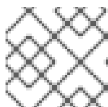
重要

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。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。

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。

6.1. CODEREADY STUDIO での JBOSS EAP XP のインストール

CodeReady Studio に JBoss EAP XP をインストールして、アプリケーションの開発に Eclipse MicroProfile 機能を使用できるようにする必要があります。



注記

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。

前提条件

- CodeReady Studio で JBoss EAP 7.3 を設定 している。
- 以下のソフトウェアアーティファクトをダウンロードししている。
  - 適切な JBoss EAP 7.3 パッチ。インストールする JBoss EAP および JBoss EAP XP の正しいパッチについては、Red Hat 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 マネージャー。
  - 適切な JBoss EAP XP パッチ。インストールする JBoss EAP および JBoss EAP XP の正しいパッチについては、Red Hat JBoss Enterprise Application Platform expansion pack Support and Life Cycle Policies ページを参照してください。

手順

1. CodeReady Studio で JBoss EAP のインストール時に指定した JBoss EAP インストールディレクトリーに移動します。

2. 先ほどダウンロードした JBoss EAP パッチを適用します。

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

以下に例を示します。

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

3. JBoss EAP の CodeReady Studio インストールで JBoss EAP XP マネージャーを設定します。

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

以下に例を示します。

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

4. ダウンロードした JBoss EAP XP パッチを適用します。

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

以下に例を示します。

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

次のステップ

MicroProfile 機能を使用するための CodeReady Studio の設定

関連情報

- JBoss EAP 製品のダウンロードページ
- Red Hat JBoss Enterprise Application Platform expansion pack サポートとライフサイクルポリシー

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

JBoss EAP で Eclipse 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 7.3 サーバーを作成します。

Eclipse 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。

前提条件

- JBoss EAP XP が CodeReady Studio にインストールされている。

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 7.3** 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 7.3 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-7.3**)。
    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、Eclipse 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 Eclipse MicroProfile クイックスタートの使用できるようになりました。

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

Eclipse 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。

以下の例は、以下の Eclipse Microprofile 機能を示しています。

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

その他のリソース

About Eclipse Microprofile

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7章 REFERENCE

7.1. ECLIPSE MICROPROFILE CONFIG リファレンス

7.1.1. デフォルトの Eclipse MicroProfile Config 属性

Eclipse 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7.1 デフォルトの Eclipse 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**

7.1.2. Eclipse MicroProfile Config SmallRye ConfigSources

microprofile-config-smallrye プロジェクトは、デフォルトの Eclipse MicroProfile Config **ConfigSource** に加えて使用できる **ConfigSource** を定義します。

表7.2 追加の Eclipse 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。Eclipse MicroProfile Config 仕様にあるデフォルトの ordinal 値は継承されます。

7.2. ECLIPSE MICROPROFILE FAULT TOLERANCE リファレンス

7.2.1. Eclipse MicroProfile Fault Tolerance 設定プロパティー

SmallRye Fault Tolerance 仕様では、Eclipse 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7.3 Eclipse 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.globalThreadPoolSize	**100**	耐障害性メカニズムによって使用されるスレッドの数。これには、バルクヘッドスレッドプールが含まれません。
io.smallrye.faulttolerance.timeoutExecutorThreads	**5**	タイムアウトのスケジューリングに使用するスレッドプールのサイズ。

7.3. ECLIPSE MICROPROFILE JWT リファレンス

7.3.1. Eclipse MicroProfile Config JWT 標準プロパティー

microprofile-jwt-smallrye サブシステムは以下の Eclipse MicroProfile Config 標準プロパティーをサポートします。

表7.4 Eclipse 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
```

7.4. ECLIPSE MICROPROFILE OPENAPI リファレンス

7.4.1. Eclipse MicroProfile OpenAPI 設定プロパティー

JBoss EAP は、標準の Eclipse MicroProfile OpenAPI 設定プロパティーに加え、以下の追加の Eclipse 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7.5 JBoss EAP の Eclipse 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。