



Red Hat JBoss Enterprise Application Platform 7.2

開発ガイド

Red Hat JBoss Enterprise Application Platform 7.2 向け

Red Hat JBoss Enterprise Application Platform 7.2 開発ガイド

Red Hat JBoss Enterprise Application Platform 7.2 向け

法律上の通知

Copyright © 2019 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

本書には、セキュアでスケーラブルな Java EE アプリケーションを迅速に開発するための手順および情報が含まれています。開発環境の設定、Maven リポジトリの使用、および開発のクラスローディングに関する知識を得ることができます。また、本書では以下を詳細に説明しています。Logging リモート JNDI ルックアップ Web アプリケーションのクラスター化 コンテキストおよび依存関係の挿入 Java Transaction API や Java Persistence API などの Java API

目次

第1章 アプリケーション開発の開始	5
1.1. JAVA ENTERPRISE EDITION 8 について	5
1.2. 開発環境の設定	5
1.3. RED HAT CODEREADY STUDIO でのアノテーション処理の設定	5
1.4. デフォルトの WELCOME WEB アプリケーションの設定	6
第2章 JBOSS EAP で MAVEN を使用	8
2.1. MAVEN について	8
2.2. MAVEN と JBOSS EAP MAVEN リポジトリのインストール	10
2.3. MAVEN リポジトリの使用	13
第3章 クラスローディングとモジュール	25
3.1. はじめに	25
3.2. デプロイメントへの明示的なモジュール依存関係の追加	26
3.3. MAVEN を使用した MANIFEST.MF エントリーの生成	30
3.4. モジュールが暗黙的にロードされないようにする	31
3.5. サブシステムをデプロイメントから除外	32
3.6. デプロイメントでのプログラムを用いたクラスローダーの使用	33
3.7. クラスローディングとサブデプロイメント	37
3.8. カスタムモードでのタグライブラリー記述子 (TLD) のデプロイ	42
3.9. クラスローディングの参照	44
第4章 LOGGING	51
4.1. ログイン	51
4.2. JBOSS LOGGING FRAMEWORK を用いたログイン	51
4.3. デプロイメントごとのログイン	54
4.4. ログインプロファイル	57
4.5. 国際化と現地語化	58
第5章 リモート JNDI ルックアップ	80
5.1. JNDI へのオブジェクトの登録	80
5.2. リモート JNDI の設定	80
5.3. HTTP 上の JNDI 呼び出し	80
第6章 WEB アプリケーションのクラスター化	82
6.1. セッションレプリケーション	82
6.2. HTTP セッションパッシベーションおよびアクティベーション	84
6.3. クラスターリングサービスのパブリック API	85
6.4. HA シングルトンサービス	86
6.5. HA シングルトンデプロイメント	89
6.6. APACHE MOD_CLUSTER-MANAGER アプリケーション	93
第7章 コンテキストおよび依存関係の挿入 (CDI)	95
7.1. CDI の概要	95
7.2. CDI を使用したアプリケーションの開発	95
7.3. あいまいな依存関係または満たされていない依存関係	99
7.4. 管理 BEAN	102
7.5. コンテキストおよびスコープ	104
7.6. 名前付き BEAN	105
7.7. BEAN ライフサイクル	106
7.8. 代替の BEAN	108
7.9. ステレオタイプ	109
7.10. オブザーバーメソッド	110

7.11. インターセプター	113
7.12. デコレーター	115
7.13. 移植可能な拡張機能	116
7.14. BEAN プロキシ	117
7.15. インジェクションでのプロキシの使用	117
第8章 JBOSS EAP MBEAN サービス	119
8.1. JBOSS MBEAN SERVICE の記述	119
8.2. JBOSS MBEAN サービスのデプロイ	121
第9章 コンカレンシーユーティリティー	122
9.1. コンテキストサービス	122
9.2. 管理対象スレッドファクトリー	123
9.3. 管理対象エグゼキューターサービス	124
9.4. 管理対象スケジュール済みエグゼキューターサービス	125
第10章 UNDERTOW	127
10.1. UNDERTOW ハンドラーについて	127
10.2. デプロイメントでの既存の UNDERTOW ハンドラーの使用	128
10.3. カスタムハンドラーの作成	129
10.4. カスタム HTTP メカニズムの開発	131
第11章 JAVA TRANSACTION API (JTA)	134
11.1. 概要	134
11.2. トランザクションの概念	134
11.3. トランザクションの最適化	142
11.4. トランザクションの結果	146
11.5. トランザクションライフサイクルの概要	147
11.6. トランザクションサブシステムの設定	148
11.7. 実際のトランザクションの使用	148
11.8. トランザクションに関するリファレンス	155
第12章 JAVA 永続 API (JPA)	158
12.1. JAVA 永続 API (JPA) について	158
12.2. 単純な JPA アプリケーションの作成	158
12.3. JPA エンティティー	162
12.4. 永続コンテキスト	162
12.5. JPA ENTITYMANAGER	163
12.6. ENTITYMANAGER の利用	164
12.7. 永続ユニットのデプロイ	165
12.8. 2次キャッシュ	166
第13章 BEAN VALIDATION	168
13.1. BEAN VALIDATION	168
13.2. バリデーション制約	168
13.3. バリデーション設定	175
第14章 WEBSOCKET アプリケーションの作成	176
WebSocket アプリケーションの作成	176
第15章 JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS)	180
15.1. JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS)	180
15.2. JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS) のセキュリティーの設定	180
第16章 JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS)	184
16.1. JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS) のセキュリティー	184

16.2. JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS) のセキュリティーの設定	184
第17章 JAVA バッチアプリケーション開発	185
17.1. 必要なバッチ依存関係	185
17.2. JOB SPECIFICATION LANGUAGE (JSL) 継承	185
17.3. バッチプロパティーインジェクション	187
第18章 クライアントの設定	192
18.1. WILDFLY-CONFIG.XML ファイルを使用したクライアント設定	192
第19章 ECLIPSE MICROPROFILE	221
19.1. ECLIPSE MICROPROFILE OPENTRACING を使用したリクエストのトレース	221
19.2. ECLIPSE MICROPROFILE HEALTH を使用したサーバー状態の監視	222
付録A リファレンス資料	224
A.1. 提供される UNDERTOW ハンドラー	224
A.2. 永続ユニットプロパティー	234
A.3. ポリシープロバイダープロパティー	235
A.4. JAVA EE 8 のプロファイルおよびテクノロジーリファレンス	236

第1章 アプリケーション開発の開始

1.1. JAVA ENTERPRISE EDITION 8 について

1.1.1. Java EE 8 プロファイルの概要

JSR 366 に定義されている Java Enterprise Edition (Java EE) 8 には、アプリケーションの特定のクラスに適した設定を表す API のサブセットであるプロファイルのサポートが含まれています。

Java EE 8 は Web および Full Platform プロファイルの仕様を定義します。Full Platform、Web Profile、または1つ以上のカスタムプロファイルのあらゆる組み合わせを実装するよう選択することができます。

- Web Profile には、web アプリケーションの開発に便利な、厳選された API のサブセットが含まれます。
- Full Platform プロファイルには、Java EE 8 の Web Profile によって定義された API と、エンタープライズアプリケーションの開発に便利な Java EE 8 API の完全セットが含まれます。

JBoss EAP 7.2 は、Java EE 8 の Full Platform および Web Profile 仕様の認定実装です。Java EE 8 の技術、JSR、および Web および Full Platform プロファイルに含まれるかどうかは、「[Java EE 8 のプロファイルおよび技術リファレンス](#)」を参照してください。

Java EE 8 API の完全リストは「[Java™ EE 8 Technologies](#)」を参照してください



注記

Java EE には、認証およびアイデンティティストアの移植可能なプラグインインターフェースを定義する [JSR 375](#) のサポートや、プログラムによるセキュリティーのアクセスポイントを提供する新しい **injectable-type SecurityContext** インターフェースも含まれています。これらの API のビルトイン実装を使用するか、カスタム実装を定義することができます。

1.2. 開発環境の設定

1. Red Hat CodeReady Studio をダウンロードしてインストールします。
手順は、Red Hat CodeReady Studio『[Installation Guide](#)』の「[Installing Developer Studio stand-alone using the Installer](#)」を参照してください。
2. Red Hat CodeReady Studio で JBoss EAP サーバーを設定します。
手順は、『[Getting Started with CodeReady Studio Tools](#)』の「[Downloading, Installing, and Setting Up JBoss EAP from within the IDE](#)」を参照してください。

1.3. RED HAT CODEREADY STUDIO でのアノテーション処理の設定

Eclipse では、アノテーション処理 (AP) はデフォルトでオフになっています。そのため、プロジェクトによって実装クラスが生成されると、**java.lang.ExceptionInInitializerError** 例外が発生した後に、プロジェクトのデプロイ時に **CLASS_NAME (implementation not found)** エラーメッセージが表示される可能性があります。

この問題は次の方法の1つで解決できます。[個別のプロジェクトのアノテーション処理を有効にする](#)か、[すべての Red Hat CodeReady Studio プロジェクトのアノテーション処理をグローバルに有効](#)にして解決します。

個別のプロジェクトのアノテーション処理を有効化

特定のプロジェクトのアノテーション処理を有効にするには、値が `jdt_apt` の `m2e.apt.activation` プロパティをプロジェクトの `pom.xml` ファイルに追加します。

```
<properties>
  <m2e.apt.activation>jdt_apt</m2e.apt.activation>
</properties>
```

この方法の例は、JBoss EAP に同梱される `logging-tools` および `kitchensink-ml` クイックスタートの `pom.xml` ファイルにあります。

Red Hat CodeReady Studio でアノテーション処理をグローバルに有効化

1. `Window` → `Preferences` の順に選択します。
2. `Maven` を展開し、`Annotation Processing` を選択します。
3. `Select Annotation Processing Mode` で `Automatically configure JDT APT (builds faster , but outcome may differ from Maven builds)` を選択し、`Apply and Close` をクリックします。

1.4. デフォルトの WELCOME WEB アプリケーションの設定

JBoss EAP には、デフォルトではポート `8080` のルートコンテキストで表示されるデフォルトの `Welcome` アプリケーションが含まれます。

このデフォルトの `Welcome` アプリケーションは、独自の Web アプリケーションで置き換えることができます。これは、以下の2つのいずれかの方法で設定できます。

- [welcome-content](#) ファイルハンドラーの変更
- [default-web-module](#) の変更

`Welcome` コンテンツを無効にすることもできます。

welcome-content ファイルハンドラーの変更

1. 新しいデプロイメントを参照する、既存の `welcome-content` ファイルハンドラーのパスを変更します。

```
/subsystem=undertow/configuration=handler/file=welcome-content:write-attribute(name=path,value="/path/to/content")
```



注記

または、サーバーのルートにより使用される異なるファイルハンドラーを作成することもできます。

```
/subsystem=undertow/configuration=handler/file=NEW_FILE_HANDLER:add(path="/path/to/content")
/subsystem=undertow/server=default-server/host=default-host/location=V:write-attribute(name=handler,value=NEW_FILE_HANDLER)
```

2. 変更を反映するためにサーバーをリロードします。

```
reload
```

default-web-module の変更

1. デプロイされた Web アプリケーションをサーバーのルートにマップします。

```
/subsystem=undertow/server=default-server/host=default-host:write-attribute(name=default-web-module,value=hello.war)
```

2. 変更を反映するためにサーバーをリロードします。

```
reload
```

デフォルトの Welcome Web アプリケーションの無効化

1. **default-host** の **location** エントリー (/) を削除して welcome アプリケーションを無効にします。

```
/subsystem=undertow/server=default-server/host=default-host/location=/: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
```

第2章 JBOSS EAP で MAVEN を使用

2.1. MAVEN について

2.1.1. Maven リポジトリ

Apache Maven は、ソフトウェアプロジェクトの作成、管理、および構築を行う Java アプリケーションの開発で使用される分散型ビルド自動化ツールです。Maven は Project Object Model (POM) と呼ばれる標準の設定ファイルを利用して、プロジェクトの定義や構築プロセスの管理を行います。POM はモジュールやコンポーネントの依存関係、ビルドの順番、結果となるプロジェクトパッケージングのターゲットを記述し、XML ファイルを使用して出力します。こうすることで、プロジェクトが正しく統一された状態で構築されるようにします。

Maven は、リポジトリを使用してアーカイブを行います。Maven リポジトリには Java ライブラリー、プラグイン、およびその他のビルドアーティファクトが格納されています。デフォルトのパブリックリポジトリは [Maven 2 Central Repository](#) ですが、複数の開発チームの間で共通のアーティファクトを共有する目的で、社内のプライベートおよび内部リポジトリとすることが可能です。また、サードパーティーのリポジトリも利用できます。JBoss EAP には、Java EE 開発者が JBoss EAP 6 でアプリケーションを構築する際に使用する要件の多くが含まれる Maven リポジトリが含まれます。このリポジトリを使用するようにプロジェクトを設定するには、「[JBoss EAP Maven リポジトリの設定](#)」を参照してください。

Maven に関する詳細は「[Welcome to Apache Maven](#)」を参照してください。

Maven リポジトリに関する詳細は、『[Apache Maven Project - Introduction to Repositories](#)』を参照してください。

2.1.2. Maven POM ファイル

Project Object Model (POM) ファイルはプロジェクトをビルドするために Maven で使用する設定ファイルです。POM ファイルは XML のファイルであり、プロジェクトの情報やビルド方法を含みます。これには、ソース、テスト、およびターゲットのディレクトリーの場所、プロジェクトの依存関係、プラグインリポジトリ、実行できるゴールが含まれます。また、バージョン、説明、開発者、メーリングリスト、ライセンスなどのプロジェクトに関する追加情報も含まれます。**pom.xml** ファイルでは一部の設定オプションを設定する必要があり、他のすべてのオプションはデフォルト値に設定されます。

pom.xml ファイルのスキーマは http://maven.apache.org/maven-v4_0_0.xsd にあります。

POM ファイルの詳細は『[Apache Maven Project POM Reference](#)』を参照してください。

Maven POM ファイルの最低要件

pom.xml ファイルの最低要件は次のとおりです。

- プロジェクトルート
- modelVersion
- groupId - プロジェクトのグループの ID
- artifactId - アーティファクト (プロジェクト) の ID
- version - 指定したグループ下のアーティファクトのバージョン

例: 基本的な **pom.xml** ファイル

基本的な **pom.xml** ファイルの例を以下に示します。

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jboss.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

2.1.3. Maven 設定ファイル

Maven の **settings.xml** ファイルには Maven のユーザー固有の設定情報が含まれています。開発者の ID、プロキシ情報、ローカルリポジリーの場所、ユーザー固有のその他の設定など、**pom.xml** ファイルで配布されてはならない情報が含まれます。

settings.xml ファイルが存在する場所は 2 つあります。

- **Maven インストール:** settings ファイルは **\$M2_HOME/conf/** ディレクトリーにあります。これらの設定は **global** 設定と呼ばれます。デフォルトの Maven 設定ファイルはコピー可能なテンプレートであり、これを基にユーザー設定ファイルを設定することが可能です。
- **ユーザーのインストール:** settings ファイルは **`\${user.home}/.m2/** ディレクトリーにあります。Maven とユーザーの **settings.xml** ファイルが両方存在する場合、内容はマージされます。重複する内容がある場合は、ユーザーの **settings.xml** ファイルが優先されます。

例 Maven 設定ファイル

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <profiles>
    <!-- Configure the JBoss EAP Maven repository -->
    <profile>
      <id>jboss-eap-maven-repository</id>
      <repositories>
        <repository>
          <id>jboss-eap</id>
          <url>file:///path/to/repo/jboss-eap-7.2.0.GA-maven-repository/maven-repository</url>
          <releases>
            <enabled>>true</enabled>
          </releases>
          <snapshots>
            <enabled>>false</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>jboss-eap-maven-plugin-repository</id>
          <url>file:///path/to/repo/jboss-eap-7.2.0.GA-maven-repository/maven-repository</url>
          <releases>
            <enabled>>true</enabled>
          </releases>
```

```

    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>
</profile>
</profiles>
<activeProfiles>
  <!-- Optionally, make the repository active by default -->
  <activeProfile>jboss-eap-maven-repository</activeProfile>
</activeProfiles>
</settings>

```

settings.xml ファイルのスキーマは <http://maven.apache.org/xsd/settings-1.0.0.xsd> にあります。

2.1.4. Maven リポジトリマネージャー

リポジトリマネージャーは、Maven リポジトリを容易に管理できるようにするツールです。リポジトリマネージャーには、次のような利点があります。

- ユーザーの組織のリポジトリとリモート Maven リポジトリとの間のプロキシを設定する機能を提供します。これには、デプロイメントの高速化や効率化、Maven によるダウンロード対象を制御するレベルの向上など、さまざまな利点があります。
- 独自に生成したアーティファクトのデプロイ先を提供し、組織内の異なる開発チーム間におけるコラボレーションを可能にします。

Maven リポジトリマネージャーに関する詳細は『[Best Practice - Using a Repository Manager](#)』を参照してください。

一般的に使用される Maven リポジトリマネージャー

Sonatype Nexus

Nexus の詳細は [Sonatype Nexus のドキュメント](#) を参照してください。

Artifactory

Artifactory の詳細は [JFrog Artifactory のドキュメント](#) を参照してください。

Apache Archiva

Apache Archiva の詳細は「[Apache Archiva: The Build Artifact Repository Manager](#)」を参照してください。



注記

リポジトリマネージャーが通常使用されるエンタープライズ環境では、Maven は、このマネージャーを使用してすべてのプロジェクトに対してすべてのアーティファクトを問い合わせる必要があります。Maven は、宣言されたすべてのリポジトリを使用して不足しているアーティファクトを見つけるため、探しているものが見つからない場合に、**central** リポジトリ (組み込みの親 POM で定義されます) で検索を試行します。この **central** の場所をオーバーライドするには、**central** で定義を追加してデフォルトの **central** リポジトリがリポジトリマネージャーになるようにします。これは、確立されたプロジェクトには適切ですが、クリーンなプロジェクトや「新しい」プロジェクトの場合は、**cyclic** 依存関係が作成されるため、問題が発生します。

2.2. MAVEN と JBOSS EAP MAVEN リポジトリのインストール

2.2.1. Maven のダウンロードおよびインストール

以下の手順に従って、Maven をダウンロードおよびインストールします。

- Red Hat CodeReady Studio を使用してアプリケーションをビルドおよびデプロイする場合は、この手順を省略します。Maven は Red Hat CodeReady Studio で配布されています。
- Maven コマンドラインを使用して JBoss EAP にアプリケーションをビルドおよびデプロイする場合は、Maven をダウンロードおよびインストールする必要があります。
 1. 「[Apache Maven Project - Download Maven](#)」にアクセスし、ご使用のオペレーティングシステムに対応する最新のディストリビューションをダウンロードします。
 2. ご使用のオペレーティングシステムに Apache Maven をダウンロードおよびインストールする方法は、Maven のドキュメントを参照してください。

2.2.2. JBoss EAP Maven リポジトリのダウンロード

以下のいずれかの方法を使用して JBoss EAP Maven リポジトリをダウンロードできます。

- [JBoss EAP Maven リポジトリの ZIP ファイルをダウンロードする](#)。
- [Offliner アプリケーションを使用して JBoss EAP Maven リポジトリをダウンロードする](#)。

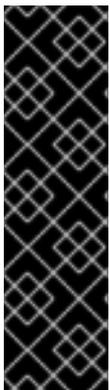
2.2.2.1. JBoss EAP Maven リポジトリの ZIP ファイルのダウンロード

以下の手順にしたがって、JBoss EAP Maven リポジトリをダウンロードします。

1. Red Hat カスタマーポータルでの [JBoss EAP ダウンロードページ](#) にログインします。
2. **Version** ドロップダウンメニューで **7.2** を選択します。
3. リストで **Red Hat JBoss Enterprise Application Platform 7.2 Maven Repository** を見つけ、**Download** をクリックしてリポジトリが含まれる ZIP ファイルをダウンロードします。
4. ZIP ファイルを希望の場所に保存します。
5. ZIP ファイルを展開します。

2.2.2.2. Offliner アプリケーションでの JBoss EAP Maven リポジトリのダウンロード

Red Hat Maven リポジトリを使用して JBoss EAP アプリケーション開発用の Maven アーティファクトをダウンロードする代わりに、Offliner アプリケーションを使用することができます。



重要

Offliner アプリケーションを使用した JBoss EAP Maven リポジトリのダウンロードプロセスは、テクノロジー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は、Red Hat カスタマーポータルでの「[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。

1. Red Hat カスタマーポータルでの [JBoss EAP ダウンロードページ](#) にログインします。
2. **Version** ドロップダウンメニューで **7.2** を選択します。
3. リストで **Red Hat JBoss Enterprise Application Platform Red Hat JBoss Enterprise Application Platform 7.2 Maven Repository Offliner Content List** を見つけ、**Download** をクリックします。
4. テキストファイルを希望のディレクトリーに保存します。



注記

このファイルにはライセンス情報が含まれません。Offliner アプリケーションによってダウンロードされたアーティファクトのライセンスは、JBoss EAP と配布される Maven リポジトリーの ZIP ファイルに指定されるライセンスと同じです。

5. Maven Central Repository から [Offliner](#) アプリケーションをダウンロードします。
6. 以下のコマンドを使用して Offliner アプリケーションを実行します。

```
$ java -jar offliner.jar -r http://repository.redhat.com/ga/ -d DOWNLOAD_FOLDER jboss-eap-7.2.0-maven-repository-content-with-sha256-checksums.txt
```

JBoss EAP Maven リポジトリーからのアーティファクトは **DOWNLOAD_FOLDER** ディレクトリーにダウンロードされます。

Offliner アプリケーションの実行の詳細は [Offliner のドキュメント](#) を参照してください。



注記

生成された JBoss EAP Maven リポジトリーの内容は、JBoss EAP Maven リポジトリーの ZIP ファイルで現在利用できる内容と同じです。Maven Central リポジトリーで利用できるアーティファクトは含まれません。

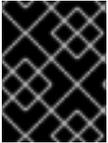
2.2.3. JBoss EAP Maven リポジトリーのインストール

JBoss EAP Maven リポジトリーはオンラインで利用でき、3つのいずれかの方法でダウンロードしてローカルにインストールすることもできます。

- JBoss EAP Maven レポジトリーをローカルファイルシステムにインストールします。詳細な手順は、[「JBoss EAP Maven リポジトリーのローカルインストール」](#) を参照してください。
- JBoss EAP Maven レポジトリーを Apache Web Server にインストールします。詳細は、[「Apache httpd で使用する JBoss EAP Maven リポジトリーのインストール」](#) を参照してください。
- JBoss EAP Maven リポジトリーを Nexus Maven リポジトリーマネージャーを使用してインストールします。詳細は、[『Repository Management Using Nexus Maven Repository Manager』](#) を参照してください。

2.2.3.1. JBoss EAP Maven リポジトリーのローカルインストール

このオプションを使用して JBoss EAP Maven リポジトリーをローカルファイルシステムにインストールします。この方法は設定が簡単で、ローカルマシンですぐに実行することができます。



重要

この方法を使用すると、開発時における Maven の使用方法を理解することができますが、チームの本番環境では推奨されません。

新しい Maven リポジトリをダウンロードする前に、`.m2/` ディレクトリーにあるキャッシュされた `repository/` サブディレクトリーを削除してください。

ローカルファイルシステムに JBoss EAP Maven リポジトリをインストールします。

1. ローカルファイルシステムに [JBoss EAP Maven リポジトリの ZIP ファイル](#) がダウンロードされていることを確認してください。
2. 希望のローカルファイルシステム上でファイルを展開します。
これにより、`maven-repository/` というサブディレクトリーに Maven リポジトリが含まれる新しい `jboss-eap-7.2.0.GA-maven-repository/` ディレクトリーが作成されます。



重要

古いローカルリポジトリを使用する場合は、そのリポジトリを Maven の `settings.xml` 設定ファイルで個別に設定する必要があります。各ローカルリポジトリは、独自の `<repository>` タグ内で設定する必要があります。

2.2.3.2. Apache httpd で使用する JBoss EAP Maven レポジトリのインストール

Web サーバーにアクセスできる開発者は Maven リポジトリにもアクセスできるため、Apache httpd で使用する JBoss EAP Maven リポジトリをインストールすることは、マルチユーザーの開発環境や複数のチームが関係する開発環境に適しています。

JBoss EAP Maven リポジトリをインストールする前に、Apache httpd を設定する必要があります。手順は、[Apache HTTP Server Project](#) ドキュメンテーションを参照してください。

1. ローカルファイルシステムに [JBoss EAP Maven リポジトリの ZIP ファイル](#) がダウンロードされていることを確認してください。
2. Apache サーバー上で Web にアクセス可能なディレクトリーに Zip 形式のファイルを展開します。
3. 作成されたディレクトリーで読み取りアクセスとディレクトリーの閲覧を許可するよう Apache を設定します。
この設定により、マルチユーザー環境が Apache httpd 上で Maven リポジトリにアクセスできるようになります。

2.3. MAVEN リポジトリの使用

2.3.1. JBoss EAP Maven リポジトリの設定

概要

プロジェクトで JBoss EAP Maven リポジトリを使用するよう Maven に指示する方法は 2 つあります。

- [リポジトリを Maven グローバルまたはユーザー設定で設定します。](#)
- [リポジトリをプロジェクトの POM ファイルで設定します。](#)

Maven 設定を使用した JBoss EAP Maven リポジトリの設定

これは推奨される方法です。リポジトリマネージャーや共有サーバー上のリポジトリを使用して 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。また、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マネージャーに特定のリポジトリ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。ミラーの詳細は、<http://maven.apache.org/guides/mini/guide-mirror-settings.html> を参照してください。

プロジェクトの POM ファイルにリポジトリ設定が含まれていない場合、この設定方法はすべての Maven プロジェクトに対して適用されます。

この項では、Maven の設定方法について説明します。Maven インストールグローバル設定またはユーザーのインストール設定を指定できます。

Maven 設定ファイルの指定

1. 使用しているオペレーションシステムの Maven **settings.xml** ファイルを見つけます。通常、このファイルは `${user.home}/.m2/` ディレクトリにあります。
 - Linux または Mac では、これは `~/.m2/` になります。
 - Windows では、これは `\Documents and Settings\.m2\` または `\Users\.m2\` になります。
2. **settings.xml** ファイルが見つからない場合は、`${user.home}/.m2/conf/` ディレクトリの **settings.xml** ファイルを `${user.home}/.m2/` ディレクトリへコピーします。
3. 以下の XML を `<profiles>` element of the **settings.xml** ファイルにコピーします。JBoss EAP リポジトリの URL を調べ、**JBOSS_EAP_REPOSITORY_URL** をその URL に置き換えます。

```

<!-- Configure the JBoss Enterprise Maven repository -->
<profile>
  <id>jboss-enterprise-maven-repository</id>
  <repositories>
    <repository>
      <id>jboss-enterprise-maven-repository</id>
      <url>JBOSS_EAP_REPOSITORY_URL</url>
      <releases>
        <enabled>>true</enabled>
      </releases>
      <snapshots>
        <enabled>>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>jboss-enterprise-maven-repository</id>
      <url>JBOSS_EAP_REPOSITORY_URL</url>
      <releases>
        <enabled>>true</enabled>
      </releases>
      <snapshots>
        <enabled>>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>

```

```

    </pluginRepository>
  </pluginRepositories>
</profile>

```

以下に、オンラインの JBoss EAP Maven リポジトリにアクセスする設定例を示します。

```

<!-- Configure the JBoss Enterprise Maven repository -->
<profile>
  <id>jboss-enterprise-maven-repository</id>
  <repositories>
    <repository>
      <id>jboss-enterprise-maven-repository</id>
      <url>https://maven.repository.redhat.com/ga/</url>
      <releases>
        <enabled>>true</enabled>
      </releases>
      <snapshots>
        <enabled>>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>jboss-enterprise-maven-repository</id>
      <url>https://maven.repository.redhat.com/ga/</url>
      <releases>
        <enabled>>true</enabled>
      </releases>
      <snapshots>
        <enabled>>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>

```

4. 次の XML を **settings.xml** ファイルの **<activeProfiles>** 要素へコピーします。

```

<activeProfile>jboss-enterprise-maven-repository</activeProfile>

```

5. Red Hat CodeReady Studio の実行中に **settings.xml** ファイルを変更する場合は、ユーザー設定を更新する必要があります。
 - a. メニューで **Window** → **Preferences** の順に選択します。
 - b. **Preferences** ウィンドウで **Maven** を展開表示し、**User Settings** を選択します。
 - c. **Update Settings** ボタンをクリックし、Red Hat CodeReady Studio で Maven のユーザー設定を更新します。



重要

Maven リポジトリに古いアーティファクトが含まれる場合は、プロジェクトをビルドまたはデプロイしたときに以下のいずれかの Maven エラーメッセージが表示されることがあります。

- Missing artifact **ARTIFACT_NAME**
- [ERROR] Failed to execute goal on project **PROJECT_NAME**; Could not resolve dependencies for **PROJECT_NAME**

この問題を解決するには、最新の Maven アーティファクトをダウンロードするためにローカルリポジトリのキャッシュバージョンを削除します。キャッシュバージョンは `${user.home}/.m2/repository/` に存在します。

プロジェクト POM を使用した JBoss EAP Maven リポジトリの設定



警告

この設定方法は、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 設定を上書きするため、回避する必要があります。

プロジェクト POM ファイルを使用してリポジトリを設定する場合は、慎重に計画する必要があります。このような設定では、推移的に含まれた POM が問題になります。これは、Maven は、外部リポジトリで不明なアーティファクトを問い合わせ、これによりビルド処理に時間がかかるようになるためです。また、アーティファクトの抽出元を制御できなくなることもあります。



注記

リポジトリの URL はリポジトリの場所 (ファイルシステムまたは Web サーバー) によって異なります。リポジトリのインストール方法は、「[JBoss EAP の Maven リポジトリのインストール](#)」を参照してください。各インストールオプションの例は次のとおりです。

ファイルシステム

`file:///path/to/repo/jboss-eap-maven-repository`

Apache Web Server

`http://intranet.acme.com/jboss-eap-maven-repository/`

Nexus リポジトリマネージャー

`https://intranet.acme.com/nexus/content/repositories/jboss-eap-maven-repository`

プロジェクトの POM ファイルの設定

1. テキストエディターでプロジェクトの `pom.xml` ファイルを開きます。
2. 次のリポジトリ設定を追加します。すでにファイルに `<repositories>` 設定が存在する場合は `<repository>` 要素を追加します。必ず `<url>` を実際のリポジトリの場所に変更するようにしてください。

```

<repositories>
  <repository>
    <id>jboss-eap-repository-group</id>
    <name>JBoss EAP Maven Repository</name>
    <url>JBOSS_EAP_REPOSITORY_URL</url>
    <layout>default</layout>
    <releases>
      <enabled>>true</enabled>
      <updatePolicy>never</updatePolicy>
    </releases>
    <snapshots>
      <enabled>>true</enabled>
      <updatePolicy>never</updatePolicy>
    </snapshots>
  </repository>
</repositories>

```

3. 次のプラグインリポジトリ設定を追加します。すでにファイルに **<pluginRepositories>** 設定が存在する場合は **<pluginRepository>** 要素を追加します。

```

<pluginRepositories>
  <pluginRepository>
    <id>jboss-eap-repository-group</id>
    <name>JBoss EAP Maven Repository</name>
    <url>JBOSS_EAP_REPOSITORY_URL</url>
    <releases>
      <enabled>>true</enabled>
    </releases>
    <snapshots>
      <enabled>>true</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>

```

JBoss EAP リポジトリの URL の確認

リポジトリの URL は、リポジトリが存在する場所によって異なります。以下のいずれかのリポジトリの場所を使用するよう Maven を設定できます。

- オンラインの JBoss EAP Maven リポジトリを使用するには、URL <https://maven.repository.redhat.com/ga/> を指定します。
- ローカルファイルシステムにインストールされた JBoss EAP Maven リポジトリを使用するには、リポジトリをダウンロードし、URL のローカルファイルパスを使用する必要があります。例: `file:///path/to/repo/jboss-eap-7.2.0.GA-maven-repository/maven-repository/`
- Apache Web Server にリポジトリをインストールする場合、リポジトリの URL は `http://intranet.acme.com/jboss-eap-7.2.0.GA-maven-repository/maven-repository/` のようになります。
- Nexus リポジトリマネージャーを使用して JBoss EAP Maven リポジトリをインストールする場合、URL は `https://intranet.acme.com/nexus/content/repositories/jboss-eap-7.2.0.GA-maven-repository/maven-repository/` のようになります。



注記

リモートリポジトリへのアクセスには、HTTP サーバーのリポジトリ用の **http://** やファイルサーバーのリポジトリ用の **file://** などの一般的なプロトコルが使用されません。

2.3.2. Red Hat CodeReady Studio で使用する Maven の設定

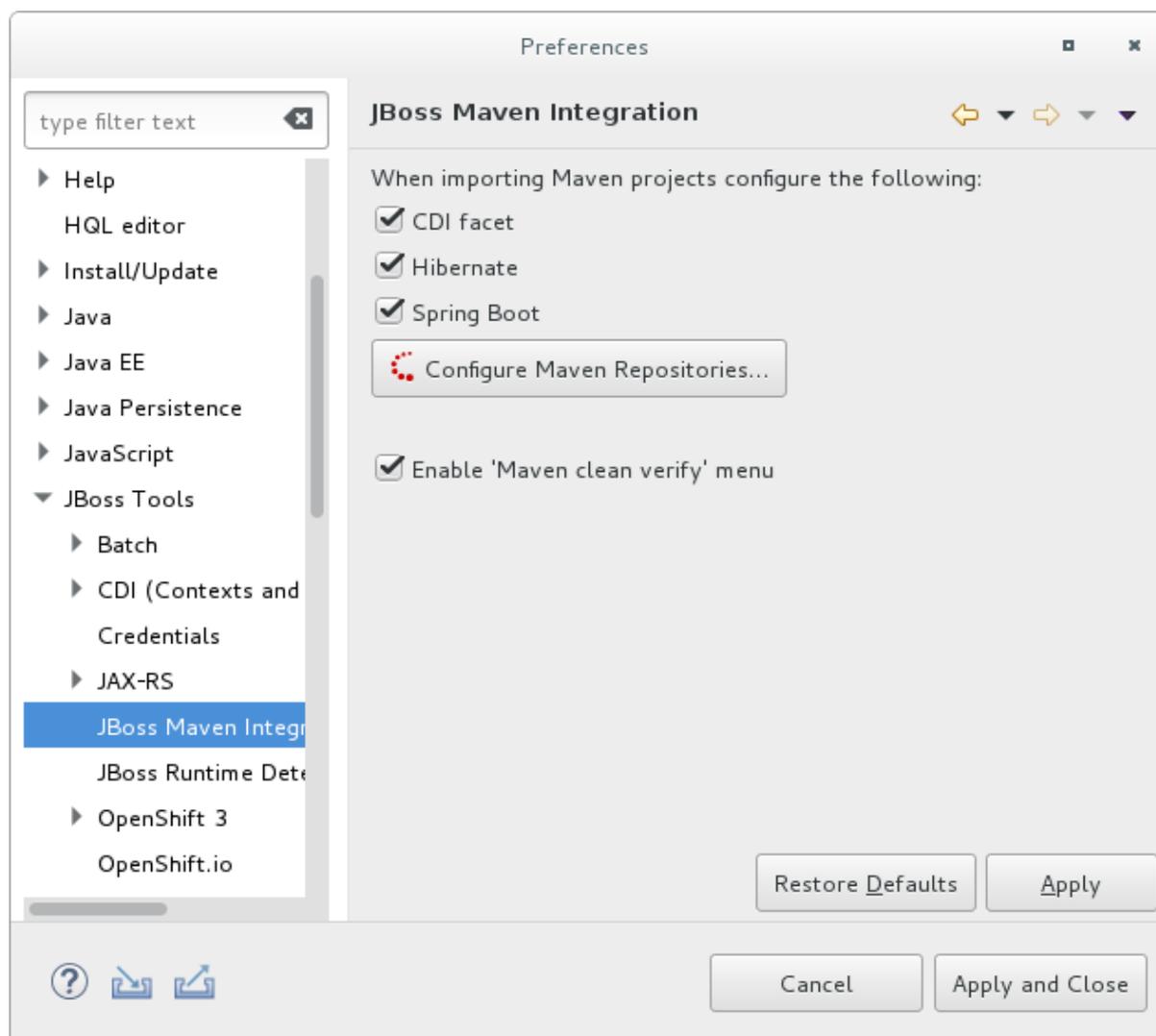
アプリケーションをビルドし、Red Hat JBoss Enterprise Application Platform にデプロイするのに必要なアーティファクトと依存関係は、パブリックリポジトリでホストされます。アプリケーションをビルドするときこのリポジトリを使用するよう Maven を設定する必要があります。ここでは、Red Hat CodeReady Studio を使用してアプリケーションをビルドおよびデプロイする場合に Maven を設定する手順について説明します。

Maven は Red Hat CodeReady Studio で配布されるため、個別にインストールする必要がありません。ただし、JBoss EAP へのデプロイメントのために Java EE Web Project ウィザードで使用する Maven を設定する必要があります。以下の手順は、Red Hat CodeReady Studio 内から Maven 設定ファイルを編集して JBoss EAP で使用する Maven を設定する方法を示しています。

Red Hat CodeReady Studio での Maven の設定

1. **Window** → **Preferences** の順にクリックし、**JBoss Tools** を展開して **JBoss Maven Integration** を選択します。

図2.1 Preferences ウィンドウの JBoss Maven 統合ペイン



2. **Configure Maven Repositories** をクリックします。
3. **Add Repository** をクリックして JBoss Enterprise Maven リポジトリを設定します。 **Add Maven Repository** ダイアログで以下の手順を実行します。
 - a. **Profile ID**、**Repository ID**、および **Repository Name** の値を **jboss-ga-repository** に設定します。
 - b. **Repository URL** の値を <http://maven.repository.redhat.com/ga> に設定します。
 - c. **Active by default** チェックボックスをクリックして Maven リポジトリを有効にします。
 - d. **OK** をクリックします。

図2.2 Maven リポジトリの追加

The screenshot shows the 'Add Maven Repository' dialog box. The title bar reads 'Add Maven Repository'. Below the title bar, there is a sub-header 'Add Maven Repository' and a Maven logo. The dialog is divided into sections: 'Profile' and 'Repository'. In the 'Profile' section, the 'Profile ID' dropdown is set to 'jboss-ga-repository' and the 'Active by default' checkbox is checked. In the 'Repository' section, the 'ID' field is 'jboss-ga-repository', the 'Name' field is 'jboss-ga-repository', and the 'URL' field is 'http://maven.repository.redhat.com/ga'. Below the 'Repository' section is an 'Advanced' section with a right-pointing arrow. At the bottom of the dialog, there is a button labeled 'Recognize JBoss Maven Enterprise Repositories...', a help icon (?), and 'Cancel' and 'OK' buttons.

4. リポジトリを確認して、**Finish** をクリックします。
5. 「Are you sure you want to update the file `MAVEN_HOME/settings.xml`?」 というメッセージが表示されます。 **Yes** をクリックして設定を更新します。 **OK** をクリックしてダイアログを閉じます。

JBoss EAP Maven リポジトリが Red Hat CodeReady Studio との使用向けに設定されます。

2.3.3. プロジェクト依存関係の管理

ここでは、Red Hat JBoss Enterprise Application Platform 向けの BOM (Bill of Materials) POM の使用方法について説明します。

BOM は、指定モジュールに対するすべてのランタイム依存関係のバージョンを指定する Maven **pom.xml** (POM) ファイルです。バージョン依存関係は、ファイルの依存関係管理セクションにリストされています。

プロジェクトは、**groupId:artifactId:version** (GAV) をプロジェクト **pom.xml** ファイルの依存関係管理セクションに追加し、**<scope>import</scope>** および **<type>pom</type>** 要素の値を指定して、BOM を使用します。



注記

多くの場合、プロジェクト POM ファイルの依存関係によって **provided** スコープが使用されます。これは、これらのクラスが実行時にアプリケーションサーバーによって提供され、ユーザーアプリケーションとともにパッケージ化する必要がないためです。

サポート対象の Maven アーティファクト

製品のビルドプロセスの一部として、JBoss EAP のすべてのランタイムコンポーネントは制御された環境でソースからビルドされます。これにより、バイナリーアーティファクトに悪意のあるコードが含まれないようにし、製品のライフサイクルが終了するまでサポートを提供できるようにします。これらのアーティファクトは、**1.0.0-redhat-1** のように使用される **-redhat** バージョン修飾子によって簡単に識別可能です。

サポートされるアーティファクトをビルド設定 **pom.xml** ファイルに追加すると、ローカルビルドおよびテスト向けの適切なバイナリーアーティファクトがビルドで使用されるようになります。**-redhat** バージョンのアーティファクトは、サポートされるパブリック API の一部とは限らず、今後の改訂で変更されることがあります。サポートされるパブリック API の詳細は、本リリースに含まれる [Javadoc ドキュメント](#) を参照してください。

たとえば、サポートされているバージョンの Hibernate を使用するには、ビルド設定に以下のようなコードを追加します。

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.3.1.Final-redhat-1</version>
  <scope>provided</scope>
</dependency>
```

上記の例には、**<version/>** の値が含まれていることに注意してください。ただし、依存関係バージョンの設定には、Maven の依存関係管理を使用することが推奨されます。

依存関係管理

Maven には、ビルド全体で直接的および推移的な依存関係のバージョンを管理するメカニズムが含まれています。依存関係管理の使用に関する一般的な情報は、Apache Maven Project の『[Introduction to the Dependency Mechanism](#)』を参照してください。

サポートされる Red Hat の依存関係を1つ以上ビルドに直接使用しても、ビルドの推移的な依存関係がすべて Red Hat アーティファクトによって完全にサポートされるとは限りません。Maven のビルドでは、Maven の中央リポジトリおよびその他の Maven リポジトリから複数のアーティファクトソー

スの組み合わせが使用することが一般的です。

JBoss EAP Maven リポジトリには、サポートされるすべての JBoss EAP バイナリーアーティファクトを指定する依存関係管理 BOM が含まれています。この BOM は、ビルドの直接的および推移的依存関係に対して、サポートされる JBoss EAP 依存関係の優先順位を決定するためにビルドで使用できます。つまり、推移的な依存関係が、サポートされる正しい依存関係バージョン (該当する場合) に対して管理されます。この BOM のバージョンは、JBoss EAP リリースのバージョンと一致します。

```
<dependencyManagement>
  <dependencies>
    ...
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>eap-runtime-artifacts</artifactId>
      <version>7.2.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>
```



注記

JBoss EAP 7 では、この BOM の名前が **eap6-supported-artifacts** から **eap-runtime-artifacts** に変更されました。この変更の目的は、この POM のアーティファクトが JBoss EAP ランタイムの一部であるが、必ずしもサポートされるパブリック API の一部ではないことを明確にすることです。一部の JAR には、リリース間で変更される可能性がある内部 API と機能が含まれます。

JBoss EAP Java EE 仕様の BOM

jboss-javaee-8.0 BOM には、JBoss EAP によって使用される Java EE 仕様の API JAR が含まれています。

プロジェクトでこの BOM を使用するには、最初に **groupId** の **org.jboss.spec** を指定して、POM ファイルの **dependencyManagement** セクションにある **jboss-javaee-8.0** BOM の依存関係を追加し、アプリケーションが必要とする特定の API の依存関係を追加します。API は **jboss-javaee-8.0** BOM に含まれているため、これらの依存関係はバージョンを必要とせず、**provided** の範囲を使用します。

以下の例は、**jboss-javaee-8.0** BOM の **1.0.0.Alpha1** を使用し、Servlet および JSP API の依存関係を追加します。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.spec</groupId>
      <artifactId>jboss-javaee-8.0</artifactId>
      <version>1.0.0.Alpha1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>
```

```

<dependencies>
  <dependency>
    <groupId>org.jboss.spec.javaax.servlet</groupId>
    <artifactId>jboss-servlet-api_4.0_spec</artifactId>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.jboss.spec.javaax.servlet.jsp</groupId>
    <artifactId>jboss-jsp-api_2.3_spec</artifactId>
    <scope>provided</scope>
  </dependency>
  ...
</dependencies>

```

注記

JBoss EAP は、ほとんどの製品コンポーネントの API に対する BOM をパッケージ化および提供します。これらの BOM の多くは、**org.jboss.bom** の **groupId** を用いて、大きな単一の **jboss-eap-javaee8** BOM にパッケージ化されます。**jboss-javaee-8.0** BOM の **groupId** は **org.jboss.spec** で、この大きな BOM に含まれます。そのため、この BOM にパッケージ化された追加の JBoss EAP 依存関係を使用する場合は1つの **jboss-eap-javaee8** BOM をプロジェクトの POM ファイルに追加でき、**jboss-javaee-8.0** およびその他の BOM 依存関係を個別に追加する必要はありません。

アプリケーション開発に利用できる JBoss EAP BOM

以下の表は、アプリケーションの開発に使用できる Maven BOM を示しています。

表2.1 JBoss BOM

BOM アーティファクト ID	ユースケース
jboss-eap-javaee8	JBoss EAP Java EE 8 API および追加の JBoss EAP API JAR をサポート。
jboss-eap-javaee8-with-spring4	jboss-eap-javaee8 および推奨される Spring 4 バージョン。
jboss-eap-javaee8-with-tools	jboss-eap-javaee8 および Arquillian などの開発ツール

注記

ほとんどのユースケースに対して使用方法を単純にするために、JBoss EAP 6 のこれらの BOM は少ない数の BOM に統合されました。Hibernate、ロギング、トランザクション、メッセージング、および他のパブリック API JAR は **jboss-eap-javaee8** BOM に含まれるようになり、各ユースケースで個別の BOM が必要なくなりました。

以下の例では、**7.2.0.GA** バージョンの **jboss-eap-javaee8** BOM が使用されています。

```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-javaee8</artifactId>

```

```

    <version>7.2.0.GA</version>
    <type>pom</type>
    <scope>import</scope>
  </dependency>
  ...
</dependencies>
</dependencyManagement>

<dependencies>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <scope>provided</scope>
</dependency>
...
</dependencies>

```

JBoss EAP クライアント BOM

クライアント BOM は、依存関係管理セクションを作成したり、依存関係を定義したりしません。クライアント BOM は他の BOM の集合体であり、リモートクライアントのユースケースに必要な依存関係のセットをパッケージ化するために使用されます。

wildfly-ejb-client-bom および **wildfly-jms-client-bom** の BOM は **jboss-eap-javaee8** BOM により管理されるため、プロジェクト依存関係でバージョンを管理する必要はありません。

以下に **wildfly-ejb-client-bom** および **wildfly-jms-client-bom** クライアント BOM 依存関係をプロジェクトに追加する方法の例を示します。

```

<dependencyManagement>
  <dependencies>
    <!-- JBoss stack of the Java EE APIs and related components. -->
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-javaee8</artifactId>
      <version>7.2.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
  ...
</dependencyManagement>

<dependencies>
<dependency>
  <groupId>org.jboss.eap</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <type>pom</type>
</dependency>
<dependency>
  <groupId>org.jboss.eap</groupId>
  <artifactId>wildfly-jms-client-bom</artifactId>
  <type>pom</type>
</dependency>
...
</dependencies>

```

Maven 依存関係および BOM POM ファイルの詳細は、『[Apache Maven Project - Introduction to the Dependency Mechanism](#)』を参照してください。

第3章 クラスローディングとモジュール

3.1. はじめに

3.1.1. クラスロードとモジュールの概要

JBoss EAP は、デプロイされたアプリケーションのクラスパスを制御するためにモジュール形式のクラスロードシステムを使用します。このシステムは、階層クラスローダーの従来のシステムよりも、柔軟性があり、より詳細に制御できます。開発者は、アプリケーションで利用可能なクラスに対して粒度の細かい制御を行い、アプリケーションサーバーで提供されるクラスを無視して独自のクラスを使用するようデプロイメントを設定できます。

モジュール形式のクラスローダーにより、すべての Java クラスはモジュールと呼ばれる論理グループに分けられます。各モジュールは、独自のクラスパスに追加されたモジュールからクラスを取得するために、他のモジュールの依存関係を定義できます。デプロイされた各 JAR および WAR ファイルはモジュールとして扱われるため、開発者はモジュール設定をアプリケーションに追加してアプリケーションのクラスパスの内容を制御できます。

3.1.2. デプロイメントでのクラスローディング

JBoss EAP では、クラスローディングを行うために、デプロイメントはすべてモジュールとして処理されます。このようなデプロイメントは動的モジュールと呼ばれます。クラスローディングの動作はデプロイメントの種類によって異なります。

WAR デプロイメント

WAR デプロイメントは1つのモジュールとして考慮されます。**WEB-INF/lib** ディレクトリーのクラスは **WEB-INF/classes** ディレクトリーにあるクラスと同じように処理されます。WAR にパッケージされているクラスはすべて、同じクラスローダーでロードされます。

EAR デプロイメント

EAR デプロイメントは複数のモジュールで構成され、以下のルールに従って定義されます。

1. EAR の **lib/** ディレクトリーは親モジュールと呼ばれる1つのモジュールです。
2. また、EAR 内の各 WAR デプロイメントは1つのモジュールです。
3. 同様に、EAR 内の EJB JAR デプロイメントも1つのモジュールです。

EAR 内の WAR および JAR デプロイメントなどのサブデプロイメントモジュールは、自動的に親モジュールに依存しますが、サブデプロイメントモジュール同士が自動的に依存するわけではありません。ただし、相互で自動的な依存関係はありません。これはサブデプロイメント分離と呼ばれ、デプロイメントごとまたはアプリケーションサーバー全体で無効にすることができます。

サブデプロイメントモジュール間の明示的な依存関係は、他のモジュールと同じ方法で追加することが可能です。

3.1.3. クラスローディングの優先順位

JBoss EAP のモジュール形式クラスローダーは、優先順位を決定してクラスローディングの競合が発生しないようにします。

デプロイメントに、パッケージとクラスの完全なリストがデプロイメントごとおよび依存関係ごとに作成されます。このリストは、クラスローディングの優先順位のルールに従って順序付けされます。実行

時にクラスをロードすると、クラスローダーはこのリストを検索し、最初に一致したものをロードします。こうすることで、デプロイメントクラスパス内の同じクラスやパッケージの複数のコピーが競合しないようになります。

クラスローダーは上から順にクラスをロードします。

1. **暗黙的な依存関係:** これらの依存関係 (JAVA EE API など) は JBoss EAP によって自動的に追加されます。これらの依存関係には一般的な機能や JBoss EAP によって提供される API が含まれるため、これらの依存関係のクラスローダー優先順位は最も高くなります。
暗黙的な各依存関係の完全な詳細は、「[暗黙的なモジュール依存関係](#)」を参照してください。
2. **明示的な依存関係:** これらの依存関係は、アプリケーションの **MANIFEST.MF** ファイルや新しいオプションの JBoss デプロイメント記述子 **jboss-deployment-structure.xml** ファイルを使用してアプリケーション設定に手動で追加されます。
明示的な依存関係の追加方法は、「[デプロイメントへの明示的なモジュール依存関係の追加](#)」を参照してください。
3. **ローカルリソース:** WAR ファイルの **WEB-INF/classes** または **WEB-INF/lib** ディレクトリー内など、デプロイメント内にパッケージ化されるクラスファイルです。
4. **デプロイメント間の依存関係:** これらは EAR デプロイメントにある他のデプロイメントに対する依存関係です。これには、EAR の **lib** ディレクトリーにあるクラスや他の EJB jar で定義されたクラスが含まれることがあります。

3.1.4. jboss-deployment-structure.xml

jboss-deployment-structure.xml ファイルは JBoss EAP のオプションのデプロイメント記述子です。このデプロイメント記述子を使用すると、デプロイメントでクラスローディングを制御できます。

このデプロイメント記述子の XML スキーマは、**EAP_HOME/docs/schema/jboss-deployment-structure-1_2.xsd** 下の製品インストールディレクトリーにあります。

このデプロイメント記述子を使用して実行できる主なタスクは次のとおりです。

- 明示的なモジュール依存関係を定義する。
- 特定の暗黙的な依存関係がロードされないようにする。
- デプロイメントのリソースより追加モジュールを定義する。
- EAR デプロイメントのサブデプロイメント分離の挙動を変更する。
- EAR のモジュールに追加のリソースルートを追加する。

3.2. デプロイメントへの明示的なモジュール依存関係の追加

明示的なモジュール依存関係をアプリケーションに追加すると、これらのモジュールのクラスをデプロイメント時にアプリケーションのクラスパスに追加することができます。



注記

JBoss EAP では、依存関係がデプロイメントに自動的に追加されます。詳細は、「[暗黙的なモジュール依存関係](#)」を参照してください。

要件

1. モジュールの依存関係を追加するソフトウェアプロジェクト。
2. 依存関係として追加するモジュールの名前を知っている必要があります。JBoss EAP に含まれる静的モジュールのリストは、「[含まれるモジュール](#)」を参照してください。モジュールが他のデプロイメントである場合は、JBoss EAP『[設定ガイド](#)』の「[動的モジュールの命名規則](#)」を参照してモジュール名を判断してください。

依存関係を設定するには、以下の2つの方法があります。

- デプロイメントの **MANIFEST.MF** ファイルにエントリーを追加します。
- **jboss-deployment-structure.xml** デプロイメント記述子にエントリーを追加します。

MANIFEST.MF への依存関係設定の追加

MANIFEST.MF ファイルの必要な依存関係エントリーを作成するよう Maven プロジェクトを設定できます。

1. **MANIFEST.MF** ファイルがプロジェクトにない場合は、この名前のファイルを作成します。Web アプリケーション (WAR) では、このファイルを **META-INF/** ディレクトリーに追加します。EJB アーカイブ (JAR) では、このファイルを **META-INF/** ディレクトリーに追加します。
2. 依存関係モジュール名をコンマで区切り、依存関係エントリーを **MANIFEST.MF** ファイルへ追加します。

```
Dependencies: org.javassist, org.apache.velocity, org antlr
```

- 依存関係をオプションにするには、依存関係エントリーのモジュール名に **optional** を付けます。

```
Dependencies: org.javassist optional, org.apache.velocity
```

- 依存関係エントリーのモジュール名に **export** を付けると、依存関係をエクスポートすることができます。

```
Dependencies: org.javassist, org.apache.velocity export
```

- **annotations** フラグは、EJB インターセプターを宣言するときなど、アノテーションのスキャン中に処理する必要があるアノテーションがモジュールの依存関係に含まれる場合に必要になります。この設定を行わないと、モジュールに宣言された EJB インターセプターをデプロイメントで使用できません。アノテーションのスキャンが関係するその他の状況でも、この設定が必要になる場合があります。

```
Dependencies: org.javassist, test.module annotations
```

- デフォルトでは、依存関係の **META-INF** 内のアイテムにはアクセスできません。**services** 依存関係により **META-INF/services** のアイテムにアクセスできるようになり、モジュール内の **services** をロードできるようになります。

```
Dependencies: org.javassist, org.hibernate services
```

- **beans.xml** ファイルをスキャンし、生成される Bean をアプリケーションが利用できるようにするために、**meta-inf** 依存関係を使用できます。

```
Dependencies: org.javassist, test.module meta-inf
```

jboss-deployment-structure.xml への依存関係設定の追加

1. アプリケーションに **jboss-deployment-structure.xml** という名前のファイルがない場合は、このファイルを新規作成し、プロジェクトに追加します。このファイルは **<jboss-deployment-structure>** がルート要素の XML ファイルです。

```
<jboss-deployment-structure>
```

```
</jboss-deployment-structure>
```

Web アプリケーション (WAR) では、このファイルを **WEB-INF/** ディレクトリーに追加します。EJB アーカイブ (JAR) では、このファイルを **META-INF/** ディレクトリーに追加します。

2. **<deployment>** 要素をドキュメントルート内に作成し、その中に **<dependencies>** 要素を作成します。
3. **<dependencies>** ノード内に各モジュールの依存関係に対するモジュール要素を追加します。 **name** 属性をモジュールの名前に設定します。

```
<module name="org.javassist" />
```

- 値が **true** のモジュールエントリーに **optional** 属性を追加することにより依存関係をオプションにすることができます。この属性のデフォルト値は **false** です。

```
<module name="org.javassist" optional="true" />
```

- 値が **true** のモジュールエントリーに **export** 属性を追加することにより依存関係をエクスポートできます。この属性のデフォルト値は **false** です。

```
<module name="org.javassist" export="true" />
```

- アノテーションのスキャン中に処理する必要があるアノテーションがモジュール依存関係に含まれる場合は、**annotations** フラグが使用されます。

```
<module name="test.module" annotations="true" />
```

- **services** 依存関係は、この依存関係にある **services** を使用するかどうか、およびどのように使用するかを指定します。デフォルト値は **none** です。この属性に **import** の値を指定することは、依存関係モジュールの **META-INF/services** パスを含むインポートフィルターリストの最後にフィルターを追加することと同じです。この属性に **export** の値を設定することは、エクスポートフィルターリストに対して同じアクションを実行することと同じです。

```
<module name="org.hibernate" services="import" />
```

- **META-INF** 依存関係は、この依存関係の **META-INF** エントリーを使用するかどうか、およびどのように使用するかを指定します。デフォルト値は **none** です。この属性に **import** の値を指定することは、依存関係モジュールの **META-INF/**** パスを含むインポートフィルターリストの最後にフィルターを追加することと同じです。この属性に **export** の値を設定することは、エクスポートフィルターリストに対して同じアクションを実行することと同じです。

```
<module name="test.module" meta-inf="import" />
```

例: 2つの依存関係を持つ `jboss-deployment-structure.xml` ファイル

```
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.javassist" />
      <module name="org.apache.velocity" export="true" />
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

JBoss EAP では、デプロイ時に、指定されたモジュールからアプリケーションのクラスパスにクラスが追加されます。

Jandex インデックスの作成

annotations フラグは Jandex インデックスが含まれるモジュールを必要とします。JBoss EAP 7.2 では、これは自動的に生成されます。しかし、自動スキャンは CPU を消費し、デプロイメント時間が長くなるため、パフォーマンスの理由上インデックスを手作業で追加することが推奨されます。

インデックスを手作業で追加するには、モジュールに追加する新しい「インデックス JAR」を作成します。Jandex JAR を使用してインデックスを構築し、新しい JAR ファイルに挿入します。現在の実装では、モジュール内部でインデックスが JAR ファイルに追加されると、スキャンは全く実行されません。

Jandex インデックスの作成:

1. インデックスを作成します。

```
java -jar modules/system/layers/base/org/jboss/jandex/main/jandex-jandex-2.0.0.Final-redhat-1.jar $JAR_FILE
```

2. 一時作業領域を作成します。

```
mkdir /tmp/META-INF
```

3. インデックスファイルを作業ディレクトリーに移動します。

```
mv $JAR_FILE.ifx /tmp/META-INF/jandex.idx
```

- a. オプション 1: インデックスを新しい JAR ファイルに含めます。

```
jar cf index.jar -C /tmp/META-INF/jandex.idx
```

JAR をモジュールディレクトリーに置き、**module.xml** を編集してリソースルートへ追加します。

- b. オプション 2: インデックスを既存の JAR に追加します。

```
java -jar /modules/org/jboss/jandex/main/jandex-1.0.3.Final-redhat-1.jar -m $JAR_FILE
```

4. アノテーションインデックスを使用するようモジュールインポートに指示し、アノテーションのスキャンでアノテーションを見つけられるようにします。

- a. オプション 1: **MANIFEST.MF** を使用してモジュールの依存関係を追加する場合
は、`Manifest-Version` を `MANIFEST.MF` の後に追加します。例を以下に示します。

は、**annotations** をセマンティック名の上に追加します。例を以下に示します。

```
Dependencies: test.module, other.module
```

上記を以下に変更します。

```
Dependencies: test.module annotations, other.module
```

- b. オプション 2: **jboss-deployment-structure.xml** を使用してモジュールの依存関係を追加する場合は、モジュールの依存関係に **annotations="true"** を追加します。



注記

静的モジュール内のクラスで定義されたアノテーション付き Java EE コンポーネントをアプリケーションで使用する場合は、アノテーションインデックスが必要です。JBoss EAP 7.2 では、静的モジュールのアノテーションインデックスは自動的に生成されるため、作成する必要がありません。ただし、**MANIFEST.MF** または **jboss-deployment-structure.xml** ファイルのいずれかに依存関係を追加して、アノテーションを使用するようモジュールインポートに指示する必要があります。

3.3. MAVEN を使用した MANIFEST.MF エントリーの生成

Maven JAR、EJB、または WAR パッケージングプラグインを使用する Maven プロジェクトでは、**Dependencies** エントリーを持つ **MANIFEST.MF** ファイルを生成することができます。この場合、依存関係の一覧は自動的に生成されず、**pom.xml** に指定された詳細が含まれる **MANIFEST.MF** ファイルのみが作成されます。

Maven を使用して **MANIFEST.MF** エントリーを生成する前に、以下のものが必要になります。

- JAR、EJB、または WAR プラグイン (**maven-jar-plugin**、**maven-ejb-plugin**、または **maven-war-plugin**) のいずれかを使用している Maven プロジェクト。
- プロジェクトのモジュール依存関係の名前を知っている必要があります。JBoss EAP に含まれる静的モジュールのリストは、「[含まれるモジュール](#)」を参照してください。モジュールが他のデプロイメントである場合は、JBoss EAP『[設定ガイド](#)』の「[動的モジュールの命名規則](#)」を参照してモジュール名を判断してください。

モジュール依存関係が含まれる MANIFEST.MF ファイルの生成

1. プロジェクトの **pom.xml** ファイルにあるパッケージングプラグイン設定に次の設定を追加します。

```
<configuration>
  <archive>
    <manifestEntries>
      <Dependencies></Dependencies>
    </manifestEntries>
  </archive>
</configuration>
```

2. モジュール依存関係のリストを **<Dependencies>** 要素に追加します。 **MANIFEST.MF** ファイルに依存関係を追加するときと同じ形式を使用します。

```
<Dependencies>org.javassist, org.apache.velocity</Dependencies>
```

ここでは、**optional** 属性と **export** 属性を使用することもできます。

```
<Dependencies>org.javassist optional, org.apache.velocity export</Dependencies>
```

3. Maven アセンブリーゴールを使用してプロジェクトをビルドします。

```
[Localhost ]$ mvn assembly:single
```

アセンブリーゴールを使用してプロジェクトをビルドすると、指定のモジュール依存関係を持つ **MANIFEST.MF** ファイルが最終アーカイブに含まれます。

例: **pom.xml** で設定されたモジュール依存関係



注記

この例は WAR プラグインの例になりますが、JAR や EJB プラグイン (maven-jar-plugin や maven-ejb-plugin) でも動作します。

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <configuration>
      <archive>
        <manifestEntries>
          <Dependencies>org.javassist, org.apache.velocity</Dependencies>
        </manifestEntries>
      </archive>
    </configuration>
  </plugin>
</plugins>
```

3.4. モジュールが暗黙的にロードされないようにする

暗黙的な依存関係がロードされないようデプロイ可能なアプリケーションを設定できます。これは、アプリケーションサーバーにより提供される暗黙的な依存関係とは異なるバージョンのライブラリーやフレームワークがアプリケーションに含まれる場合に役に立つことがあります。

要件

- 暗黙的な依存関係を除外するソフトウェアプロジェクト。
- 除外するモジュール名を知っている必要があります。暗黙的な依存関係のリストや状態は「[暗黙的なモジュール依存関係](#)」を参照してください。

jboss-deployment-structure.xml への依存関係除外設定の追加

1. アプリケーションに **jboss-deployment-structure.xml** という名前のファイルがない場合は、このファイルを新規作成し、プロジェクトに追加します。このファイルは **<jboss-deployment-structure>** がルート要素の XML ファイルです。

```
<jboss-deployment-structure>
```

```
</jboss-deployment-structure>
```

Web アプリケーション (WAR) では、このファイルを **WEB-INF/** ディレクトリーに追加します。EJB アーカイブ (JAR) では、このファイルを **META-INF/** ディレクトリーに追加します。

2. **<deployment>** 要素をドキュメントルート内に作成し、その中に **<exclusions>** 要素を作成します。

```
<deployment>
```

```
<exclusions>
```

```
</exclusions>
```

```
</deployment>
```

3. **exclusions** 要素内で、除外する各モジュールに対して **<module>** 要素を追加します。 **name** 属性をモジュールの名前に設定します。

```
<module name="org.javassist" />
```

例: 2 つのモジュールの除外

```
<jboss-deployment-structure>
```

```
<deployment>
```

```
<exclusions>
```

```
<module name="org.javassist" />
```

```
<module name="org.dom4j" />
```

```
</exclusions>
```

```
</deployment>
```

```
</jboss-deployment-structure>
```

3.5. サブシステムをデプロイメントから除外

サブシステムの除外は、サブシステムの削除と同じ効果がありますが、単一のデプロイメントにのみ適用されます。 **jboss-deployment-structure.xml** 設定ファイルを編集することにより、デプロイメントからサブシステムを除外できます。

サブシステムの除外

1. **jboss-deployment-structure.xml** ファイルを編集します。
2. **<deployment>** タグ内に以下の XML を追加します。

```
<exclude-subsystems>
```

```
<subsystem name="SUBSYSTEM_NAME" />
```

```
</exclude-subsystems>
```

3. **jboss-deployment-structure.xml** ファイルを保存します。

サブシステムのデプロイメントユニットプロセッサがデプロイメント上で実行されなくなります。

例: **jboss-deployment-structure.xml** ファイル

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.2">
  <ear-subdeployments-isolated>true</ear-subdeployments-isolated>
  <deployment>
    <exclude-subsystems>
      <subsystem name="jaxrs" />
    </exclude-subsystems>
    <exclusions>
      <module name="org.javassist" />
    </exclusions>
    <dependencies>
      <module name="deployment.javassist.proxy" />
      <module name="deployment.myjavassist" />
      <module name="myservicemodule" services="import"/>
    </dependencies>
    <resources>
      <resource-root path="my-library.jar" />
    </resources>
  </deployment>
  <sub-deployment name="myapp.war">
    <dependencies>
      <module name="deployment.myear.ear.myejbjar.jar" />
    </dependencies>
    <local-last value="true" />
  </sub-deployment>
  <module name="deployment.myjavassist" >
    <resources>
      <resource-root path="javassist.jar" >
        <filter>
          <exclude path="javassist/util/proxy" />
        </filter>
      </resource-root>
    </resources>
  </module>
  <module name="deployment.javassist.proxy" >
    <dependencies>
      <module name="org.javassist" >
        <imports>
          <include path="javassist/util/proxy" />
          <exclude path="/*" />
        </imports>
      </module>
    </dependencies>
  </module>
</jboss-deployment-structure>
```

3.6. デプロイメントでのプログラムを用いたクラスローダーの使用

3.6.1. デプロイメントでのプログラムによるクラスおよびリソースのロード

プログラムを用いて、アプリケーションコードでクラスやリソースを検索またはロードできます。複数の要素に応じてその方法を選択します。ここでは、使用できる方法を説明し、それらの方法を使用するガイドラインを提供します。

Class.forName() メソッドを使用したクラスのロード

Class.forName() メソッドを使用すると、プログラムでクラスをロードおよび初期化できます。このメソッドには2つのシグネチャーがあります。

- **Class.forName(String className):**
このシグネチャーは、1つのパラメーター (ロードする必要があるクラスの名前) のみを取ります。このメソッドシグネチャーを使用すると、現在のクラスのクラスローダーによってクラスがロードされ、デフォルトで新たにロードされたクラスが初期化されます。
- **Class.forName(String className, boolean initialize, ClassLoader loader):**
このシグネチャーは、クラス名、クラスを初期化するかどうかを指定するブール値、およびクラスをロードする **ClassLoader** の3つのパラメーターを想定します。

プログラムでクラスをロードする場合は、3つの引数のシグネチャーを用いる方法が推奨されます。このシグネチャーを使用すると、ロード時に目的のクラスを初期化するかどうかを制御できます。また、JVM はコールスタックをチェックして、使用するクラスローダーを判断する必要がないため、クラスローダーの取得および提供がより効率的になります。コードが含まれるクラスの名前が **CurrentClass** である場合は、**CurrentClass.class.getClassLoader()** メソッドを使用してクラスのクラスローダーを取得できます。

以下は、ロードするクラスローダーを提供し、**TargetClass** クラスを初期化する例になります。

```
Class<?> targetClass = Class.forName("com.myorg.util.TargetClass", true,
CurrentClass.class.getClassLoader());
```

名前ですべてのリソースを検索

リソースの名前とパスが分かり、直接そのリソースをロードする場合は、標準的な Java Development Kit (JDK) の **Class** または **ClassLoader** API を使用するのが最良の方法です。

- 単一のリソースをロードします。
ご使用のクラスと同じディレクトリーまたはデプロイメントの他のクラスと同じディレクトリーにある単一のリソースをロードする場合は、**Class.getResourceAsStream()** メソッドを使用できます。

```
InputStream inputStream =
CurrentClass.class.getResourceAsStream("targetResourceName");
```

- 単一リソースのインスタンスをすべてロードします。
デプロイメントのクラスローダーが認識できる単一リソースのすべてのインスタンスをロードするには、**Class.getClassLoader().getResources(String resourceName)** メソッドを使用します。ここで、**resourceName** はリソースの完全修飾パスに置き換えます。このメソッドは、指定の名前でクラスローダーがアクセスできるリソースに対し、すべての **URL** オブジェクトの列挙を返します。その後、URL の配列で繰り返し処理し、**openStream()** メソッドを使用して各ストリームを開くことができます。

以下の例は、1つのリソースのすべてのインスタンスをロードし、結果で処理を繰り返します。

```
Enumeration<URL> urls =
CurrentClass.class.getClassLoader().getResources("full/path/to/resource");
while (urls.hasMoreElements()) {
    URL url = urls.nextElement();
    InputStream inputStream = null;
    try {
        inputStream = url.openStream();
        // Process the inputStream
    }
```

```

...
} catch(IOException ioException) {
    // Handle the error
} finally {
    if (inputStream != null) {
        try {
            inputStream.close();
        } catch (Exception e) {
            // ignore
        }
    }
}
}
}

```



注記

URL インスタンスはローカルストレージからロードされるため、**openConnection()** や他の関連するメソッドを使用する必要はありません。ストリームは非常に簡単に使用でき、ストリームを使用することにより、コードの複雑さが最小限に抑えられます。

- クラスローダーからクラスファイルをロードします。クラスがすでにロードされている場合は、以下の構文を使用して、そのクラスに対応するクラスファイルをロードできます。

```

InputStream inputStream =
    CurrentClass.class.getResourceAsStream(TargetClass.class.getSimpleName() + ".class");

```

クラスがロードされていない場合は、クラスローダーを使用し、パスを変換する必要があります。

```

String className = "com.myorg.util.TargetClass"
InputStream inputStream =
    CurrentClass.class.getClassLoader().getResourceAsStream(className.replace('.', '/') +
".class");

```

3.6.2. デプロイメントでのプログラムによるリソースの繰り返し

JBoss Modules ライブラリーは、すべてのデプロイメントリソースを繰り返し処理するために複数の API を提供します。JBoss Modules API の JavaDoc は <http://docs.jboss.org/jbossmodules/1.3.0.Final/api/> にあります。これらの API を使用するには、以下の依存関係を **MANIFEST.MF** に追加する必要があります。

```
Dependencies: org.jboss.modules
```

これらの API により柔軟性が向上しますが、直接パスを検索するよりも動作がかなり遅くなることに注意してください。

ここでは、アプリケーションコードでプログラムを用いてリソースを繰り返す方法を説明します。

- デプロイメント内およびすべてのインポート内のリソースをリストします。場合によっては、正確なパスでリソースを検索できないことがあります。たとえば、正確なパスがわからなかったり、指定のパスで複数のファイルをチェックする必要がある場合などで

す。このような場合、JBoss Modules ライブラリーはすべてのデプロイメントを繰り返し処理するための API を複数提供します。2つのメソッドのいずれかを使用すると、デプロイメントでリソースを繰り返し処理できます。

- 単一のモジュールで見つかったすべてのリソースを繰り返し処理します。

ModuleClassLoader.iterateResources() メソッドは、このモジュールクラスローダー内のすべてのリソースを繰り返し処理します。このメソッドは、検索を開始するディレクトリーの名前と、サブディレクトリーで再帰的に処理するかどうかを指定するブール値の2つの引数を取ります。

以下の例は、ModuleClassLoader の取得方法と、**bin/** ディレクトリーにあるリソースのイテレーターの取得方法 (サブディレクトリーを再帰的に検索) を示しています。

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources = moduleClassLoader.iterateResources("bin",true);
```

取得されたイテレーターは、一致した各リソースをチェックし、名前とサイズのクエリー (可能な場合) を行うために使用できます。また、読み取り可能ストリームを開いたり、リソースの URL を取得するために使用できます。

- 単一のモジュールで見つかったすべてのリソースとインポートされたリソースを繰り返し処理します。

Module.iterateResources() メソッドは、このモジュールクラスローダー内のすべてのリソース (モジュールにインポートされたリソースを含む) を繰り返し処理します。このメソッドは、前述のメソッドよりもはるかに大きなセットを返します。このメソッドには、特定パターンの結果を絞り込むフィルターとなる引数が必要になります。代わりに、**PathFilters.acceptAll()** を指定してセット全体を返すことも可能です。

以下の例は、インポートを含む、このモジュールのリソースのセット全体を検索する方法を示しています。

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Module module = moduleClassLoader.getModule();
Iterator<Resource> moduleResources =
module.iterateResources(PathFilters.acceptAll());
```

- パターンと一致するすべてのリソースを検索します。

デプロイメント内またはデプロイメントの完全なインポートセット内で特定のリソースのみを見つける必要がある場合は、リソースの繰り返しをフィルターする必要があります。JBoss Modules のフィルター API は、リソースの繰り返しをフィルターする複数のツールを提供します。

- 依存関係の完全セットを確認します。
依存関係の完全なセットをチェックする必要がある場合は、**Module.iterateResources()** メソッドの **PathFilter** パラメーターを使用して、一致する各リソースの名前を確認できます。
- デプロイメント依存関係を確認します。
デプロイメント内のみを検索する必要がある場合は、**ModuleClassLoader.iterateResources()** メソッドを使用します。が、追加のメソッドを使用して結果となるイテレーターをフィルターする必要があります。**PathFilters.filtered()** メソッドは、リソースイテレーターのフィルターされたビューを

提供できます。**PathFilters** クラスには、さまざまな関数を実行するフィルターを作成する多くの静的メソッドが含まれています。これには、子パスや完全一致の検索、Ant 形式の「glob」パターンの一致などが含まれます。

- リソースのフィルターに関する追加のコード例。
以下の例は、異なる基準を基にしてリソースをフィルターする方法を示しています。

例: デプロイメントでファイル名が `messages.properties` のファイルをすべて検索

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources =
PathFilters.filtered(PathFilters.match("**/messages.properties"),
moduleClassLoader.iterateResources("", true));
```

例: デプロイメントおよびインポートでファイル名が `messages.properties` のファイルをすべて検索

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Module module = moduleClassLoader.getModule();
Iterator<Resource> moduleResources =
module.iterateResources(PathFilters.match("**/message.properties"));
```

例: デプロイメントでディレクトリー名が `my-resources` であるディレクトリー内部のファイルをすべて検索

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources = PathFilters.filtered(PathFilters.match("**/my-
resources/**"), moduleClassLoader.iterateResources("", true));
```

例: デプロイメントおよびインポートで `messages` または `errors` という名前のファイルをすべて検索

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Module module = moduleClassLoader.getModule();
Iterator<Resource> moduleResources =
module.iterateResources(PathFilters.any(PathFilters.match("**/messages"),
PathFilters.match("**/errors")));
```

例: デプロイメントで指定のパッケージにあるすべてのファイルを検索

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources =
moduleClassLoader.iterateResources("path/form/of/packageName", false);
```

3.7. クラスローディングとサブデプロイメント

3.7.1. エンタープライズアーカイブのモジュールおよびクラスロード

エンタープライズアーカイブ (EAR) は、JAR または WAR デプロイメントのように、単一モジュールとしてロードされません。これらは、複数の一意のモジュールとしてロードされます。

以下のルールによって、EAR に存在するモジュールが決定されます。

- EAR アーカイブのルートにある **lib/** ディレクトリーの内容はモジュールです。これは、親モジュールと呼ばれます。
- 各 WAR および EJB JAR サブデプロイメントはモジュールです。これらのモジュールの動作は、他のモジュールおよび親モジュールの暗黙的な依存関係と同じです。
- サブデプロイメントでは、親モジュールとすべての他の非 WAR サブデプロイメントに暗黙的な依存関係が存在します。

JBoss EAP ではサブデプロイメントクラスローダーの分離がデフォルトで無効になっているため、WAR サブデプロイメント以外の暗黙的な依存関係が発生します。親モジュールの依存関係は、サブデプロイメントクラスローダーの分離に関係なく永続します。



重要

サブデプロイメントでは、WAR サブデプロイメントに暗黙的な依存関係が存在しません。他のモジュールと同様に、サブデプロイメントは、別のサブデプロイメントの明示的な依存関係で設定できます。

サブデプロイメントクラスローダーの分離は、厳密な互換性が必要な場合に有効にできます。これは、単一の EAR デプロイメントまたはすべての EAR デプロイメントに対して有効にできます。Java EE 6 の仕様では、依存関係が各サブデプロイメントの **MANIFEST.MF** ファイルの **Class-Path** エントリーとして明示的に宣言されている場合を除き、移植可能なアプリケーションがお互いにアクセスできるサブデプロイメントに依存しないことが推奨されます。

3.7.2. サブデプロイメントクラスローダーの分離

エンタープライズアーカイブ (EAR) の各サブデプロイメントは独自のクラスローダーを持つ動的モジュールです。デフォルトでは、サブデプロイメントは他のサブデプロイメントのリソースにアクセスできません。

サブデプロイメントが他のサブデプロイメントのリソースにアクセスすることが許可されていない場合は、厳格なサブデプロイメントの分離を有効にできます。

3.7.3. EAR 内のサブデプロイメントクラスローダーの分離を有効にする

このタスクでは、EAR の特別なデプロイメント記述子を使用して EAR デプロイメントのサブデプロイメントクラスローダーの分離を有効にする方法を示します。アプリケーションサーバーを変更する必要はなく、他のデプロイメントは影響を受けません。



重要

サブデプロイメントクラスローダーの分離が無効であっても、WAR を依存関係として追加することはできません。

1. デプロイメント記述子ファイルを追加します。

jboss-deployment-structure.xml デプロイメント記述子ファイルが EAR の **META-INF** ディレクトリーに存在しない場合は追加し、次の内容を追加します。

```
<jboss-deployment-structure>
</jboss-deployment-structure>
```

2. **<ear-subdeployments-isolated>** 要素を追加します。
<ear-subdeployments-isolated> 要素が **jboss-deployment-structure.xml** ファイルに存在しない場合は追加し、内容が **true** になるようにします。

```
<ear-subdeployments-isolated>true</ear-subdeployments-isolated>
```

この EAR デプロイメントに対してサブデプロイメントクラスローダーの分離が有効になります。つまり、EAR のサブデプロイメントは WAR ではないサブデプロイメントごとに自動的な依存関係を持ちません。

3.7.4. エンタープライズアーカイブのサブデプロイメント間で共有するセッションの設定

JBoss EAP では、EAR に含まれる WAR モジュールサブデプロイメント間でセッションを共有するようエンタープライズアーカイブ (EAR) を設定する機能が提供されます。この機能はデフォルトで無効になり、EAR の **META-INF/jboss-all.xml** ファイルで明示的に有効にする必要があります。



重要

この機能は標準的サーブレット機能ではないため、この機能が有効な場合はアプリケーションを移植できないことがあります。

EAR 内の WAR 間で共有するセッションを有効にするには、EAR の **META-INF/jboss-all.xml** で **shared-session-config** 要素を宣言する必要があります。

例: **META-INF/jboss-all.xml**

```
<jboss xmlns="urn:jboss:1.0">
...
<shared-session-config xmlns="urn:jboss:shared-session-config:1.0">
</shared-session-config>
...
</jboss>
```

shared-session-config 要素は、EAR 内のすべての WAR に対して共有セッションマネージャーを設定するために使用されます。**shared-session-config** 要素が存在する場合は、EAR 内のすべての WAR で同じセッションマネージャーが共有されます。ここで行われる変更は、EAR 内に含まれるすべての WAR に影響します。

3.7.4.1. 共有セッション設定オプションのリファレンス

例: **META-INF/jboss-all.xml**

```
<jboss xmlns="urn:jboss:1.0">
<shared-session-config xmlns="urn:jboss:shared-session-config:1.0">
```

```

<max-active-sessions>10</max-active-sessions>
<session-config>
  <session-timeout>0</session-timeout>
  <cookie-config>
    <name>JSESSIONID</name>
    <domain>domainName</domain>
    <path>/cookiePath</path>
    <comment>cookie comment</comment>
    <http-only>true</http-only>
    <secure>true</secure>
    <max-age>1</max-age>
  </cookie-config>
  <tracking-mode>COOKIE</tracking-mode>
</session-config>
<replication-config>
  <cache-name>web</cache-name>
  <replication-granularity>SESSION</replication-granularity>
</replication-config>
</shared-session-config>
</jboss>

```

要素	説明
shared-session-config	共有セッション設定のルート要素。この要素が META-INF/jboss-all.xml に存在しない場合は、EAR に含まれるすべてのデプロイ済み WAR で単一のセッションマネージャーが共有されます。
max-active-sessions	許可される最大セッション数。
session-config	EAR に含まれるすべてのデプロイ済み WAR に対するセッション設定パラメーターを含みます。
session-timeout	EAR に含まれるデプロイ済み WAR で作成されたすべてのセッションに対するデフォルトのセッションタイムアウト間隔を定義します。指定されたタイムアウトは、分単位の整数で表記する必要があります。タイムアウトが 0 またはそれよりも小さい値である場合は、コンテナにより、セッションのデフォルトの動作がタイムアウトしなくなります。この要素が指定されない場合は、コンテナでデフォルトのタイムアウト期間を設定する必要があります。
cookie-config	EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーを含みます。
name	EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーに割り当てられる名前。デフォルト値は JSESSIONID です。
domain	EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーに割り当てられるドメイン名。

要素	説明
path	EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーに割り当てられるパス。
comment	EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーに割り当てられるコメント。
http-only	EAR に含まれるデプロイ済みの WAR によって作成されたセッション追跡クッキーを HttpOnly とマークするかどうかを指定します。
secure	対応するセッションを開始したリクエストが HTTPS ではなくプレーン HTTP を使用している場合であっても、EAR に含まれるデプロイ済みの WAR によって作成されたセッション追跡クッキーをセキュアとマークするかどうかを指定します。
max-age	EAR に含まれるデプロイ済みの WAR によって作成されたセッション追跡クッキーに割り当てられる有効期間 (秒単位)。デフォルト値は -1 です。
tracking-mode	EAR に含まれるデプロイ済み WAR により作成されたセッションの追跡モードを定義します。
replication-config	HTTP セッションクラスタリング設定を含みます。
cache-name	このオプションはクラスタリング専用です。セッションデータを格納する Infinispan コンテナとキャッシュの名前を指定します。デフォルト値が明示的に設定されていない場合は、アプリケーションサーバーによってデフォルト値が決定されます。キャッシュコンテナ内で特定のキャッシュを使用するには、 web.dist のように container.cache という形式を使用します。名前が修飾されていない場合は、指定されたコンテナのデフォルトのキャッシュが使用されます。

要素	説明
replication-granularity	<p>このオプションはクラスタリング専用です。セッションレプリケーションの粒度を決定します。可能な値は SESSION と ATTRIBUTE で、デフォルト値は SESSION です。</p> <p>SESSION 粒度が使用される場合は、すべてのセッション属性がレプリケートされます (要求の範囲内でいずれかのセッション属性が変更された場合)。このポリシーは、オブジェクト参照が複数のセッション属性で共有される場合に必要です。ただし、セッション属性が非常に大きい場合や頻繁に変更されない場合は非効率になることがあります。これは、属性が変更されたかどうかに関係なく、すべての属性をレプリケートする必要があるためです。</p> <p>ATTRIBUTE 粒度が使用される場合は、要求の範囲内で変更された属性のみがレプリケートされます。オブジェクト参照が複数のセッション属性で共有される場合、このポリシーは適切ではありません。セッション属性が非常に大きい場合や頻繁に変更されない場合は SESSION よりも効率的になることがあります。</p>

3.8. カスタムモードでのタグライブラリー記述子 (TLD) のデプロイ

共通のタグライブラリー記述子 (TLD) を使用する複数のアプリケーションがある場合、アプリケーションから TLD を分離し、一元的で一意的な場所に置くと有用であることがあります。これにより、TLD を使用するアプリケーションごとに更新を行う必要がなくなり、TLD への追加や更新が簡単になります。

これを行うには、TLD JAR が含まれるカスタム JBoss EAP モジュールを作成し、アプリケーションでそのモジュールの依存関係を宣言します。



注記

少なくとも1つの JAR に TLD が含まれ、TLD が **META-INF** に含まれるようにします。

カスタムモジュールでの TLD のデプロイ

1. 管理 CLI を使用して、JBoss EAP インスタンスへ接続し、以下のコマンドを実行して TLD JAR が含まれるカスタムモジュールを作成します。

```
module add --name=MyTagLibs --resources=/path/to/TLArchive.jar
```

重要

module 管理 CLI コマンドを使用したモジュールの追加および削除は、テクノロジープレビューとしてのみ提供されます。このコマンドは、管理対象ドメインでの使用や、リモートによる管理 CLI への接続時には適していません。本番環境では、モジュールを手作業で追加および削除する必要があります。詳細は、JBoss EAP『[設定ガイド](#)』の「[カスタムモジュールの手動作成](#)」および「[手作業によるカスタムモジュールの削除](#)」を参照してください。

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は、Red Hat カスタマーポータル「[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。

TLD が依存関係を必要とするクラスとともにパッケージ化されている場合は、**--dependencies** オプションを使用して、カスタムモジュールの作成時にこれらの依存関係を指定するようにします。

モジュールを作成するときに、システムのファイルシステム固有の区切り文字を使用して複数の JAR リソースを指定できます。

- Linux の場合 - :例: **--resources=<path-to-jar>:<path-to-another-jar>**
- Windows の場合 - ;例: **--resources=<path-to-jar>;<path-to-another-jar>**

注記

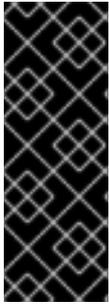
--resources

これは **--module-xml** を使用しない限り必要です。ファイルシステム固有のパス区切り文字 (たとえば、`java.io.File.pathSeparatorChar`) で区切られたファイルシステムパス (通常は JAR ファイル) をリストします。指定されたファイルは作成されたモジュールのディレクトリーにコピーされます。

--resource-delimiter

これは、リソース引数のオプションのユーザー定義パス区切り文字です。この引数が存在する場合、コマンドパーサーはファイルシステム固有のパス区切り文字の代わりにその値を使用します。これにより、**modules** コマンドをクロスプラットフォームのスクリプトで使用できるようになります。

2. ご使用のアプリケーションで、[デプロイメントへの明示的なモジュール依存関係の追加](#)で説明されているいずれかの方法を使用して新しい MyTagLibs カスタムモジュールの依存関係を宣言します。



重要

依存関係を宣言するときは必ず **META-INF** もインポートしてください。たとえば、**MANIFEST.MF** の場合は以下ようになります。

```
Dependencies: com.MyTagLibs meta-inf
```

jboss-deployment-structure.xml の場合は、**meta-inf** 属性を使用してください。

3.9. クラスローディングの参照

3.9.1. 暗黙的なモジュール依存関係

以下の表には、依存関係としてデプロイメントに自動的に追加されるモジュールと、依存関係をトリガーする条件が記載されています。

表3.1暗黙的なモジュール依存関係

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
Application Client	<ul style="list-style-type: none"> ● org.omg.api ● org.jboss.xnio 		
Batch	<ul style="list-style-type: none"> ● javax.batch.api ● org.jberet.jberet-core ● org.wildfly.jberet 		
Bean Validation	<ul style="list-style-type: none"> ● org.hibernate.validator ● javax.validation.api 		
Core Server	<ul style="list-style-type: none"> ● javax.api ● sun.jdk ● org.jboss.vfs ● ibm.jdk 		
DriverDependenciesProcessor		<ul style="list-style-type: none"> ● javax.transaction.api 	

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
EE	<ul style="list-style-type: none"> ● org.jboss.invocation (org.jboss.invocation.proxy.classloading を除く) ● org.jboss.as.ee (org.jboss.as.ee.component.serialization、org.jboss.as.ee.concurrent、および org.jboss.as.ee.concurrent.handle を除く) ● org.wildfly.naming ● javax.annotation.api ● javax.enterprise.concurrent.api ● javax.interceptor.api ● javax.json.api ● javax.resource.api ● javax.rmi.api ● javax.xml.bind.api ● javax.api ● org.glassfish.javax.el ● org.glassfish.javax.enterprise.concurrent 		
EJB 3	<ul style="list-style-type: none"> ● javax.ejb.api ● javax.xml.rpc.api ● org.jboss.ejb-client ● org.jboss.jiio-client ● org.jboss.as.ejb3 	<ul style="list-style-type: none"> ● org.wildfly.jiio-openjdk 	

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
IIOP	<ul style="list-style-type: none"> ● org.omg.api ● javax.rmi.api ● javax.orb.api 		
JAX-RS (RESTEasy)	<ul style="list-style-type: none"> ● javax.xml.bind.api ● javax.ws.rs.api ● javax.json.api ● org.jboss.resteasy.resteasy-atom-provider ● org.jboss.resteasy.resteasy-crypto ● org.jboss.resteasy.resteasy-validator-provider-11 ● org.jboss.resteasy.resteasy-jaxrs ● org.jboss.resteasy.resteasy-jaxb-provider ● org.jboss.resteasy.resteasy-jackson2-provider ● org.jboss.resteasy.resteasy-jsapi ● org.jboss.resteasy.resteasy-json-p-provider ● org.jboss.resteasy.resteasy-multipart-provider ● org.jboss.resteasy.resteasy-yaml-provider ● org.codehaus.jackson-jackson-core-asl 	<ul style="list-style-type: none"> ● org.jboss.resteasy.resteasy-cdi 	<p>デプロイメントに JAX-RS アノテーションが存在すること。</p>

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
JCA	<ul style="list-style-type: none"> ● javax.resource.api 	<ul style="list-style-type: none"> ● javax.jms.api ● javax.validation.api ● org.jboss.ironjacamar.api ● org.jboss.ironjacamar.impl ● org.hibernate.validator 	リソースアダプター (RAR) アーカイブのデプロイメント。
JPA (Hibernate)	<ul style="list-style-type: none"> ● javax.persistence.api 	<ul style="list-style-type: none"> ● org.jboss.as.jpa ● org.jboss.as.jpa.spi ● org.javassist 	<p>@PersistenceUnit または @PersistenceContext アノテーションが存在するか、デプロイメント記述子に <persistence-unit-ref> または <persistence-context-ref> 要素が存在すること。</p> <p>JBoss EAP は永続プロバイダー名をモジュール名にマップします。 persistence.xml ファイルで特定のプロバイダーに名前を付けると、適切なモジュールに対して依存関係が追加されます。これが希望の挙動ではない場合は、 jboss-deployment-structure.xml を使用して除外できます。</p>
JSF (Java Server Faces)		<ul style="list-style-type: none"> ● javax.faces.api ● com.sun.jsf-impl ● org.jboss.as.jsf ● org.jboss.as.jsf-injection 	<p>EAR アプリケーションに追加されます。</p> <p>値が true の org.jboss.jbossfaces.WAR_BUNDLES_JSF_IMPL の context-param を web.xml ファイルで指定しない場合のみ WAR アプリケーションに追加されず。</p>
JSR-77	<ul style="list-style-type: none"> ● javax.management.j2ee.api 		

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
Logging	<ul style="list-style-type: none"> ● org.jboss.logging ● org.apache.commons.logging ● org.apache.log4j ● org.slf4j ● org.jboss.logging.jul-to-slf4j-stub 		
Mail	<ul style="list-style-type: none"> ● javax.mail.api ● javax.activation.api 		
Messaging	<ul style="list-style-type: none"> ● javax.jms.api 	<ul style="list-style-type: none"> ● org.wildfly.extension.messaging-activemq 	
PicketLink Federation		<ul style="list-style-type: none"> ● org.picketlink 	
Pojo	<ul style="list-style-type: none"> ● org.jboss.as.pojo 		
SAR		<ul style="list-style-type: none"> ● org.jboss.modules ● org.jboss.as.system-jmx ● org.jboss.common-beans 	jboss-service.xml を含む SAR アーカイブのデプロイメント。
Seam2		<ul style="list-style-type: none"> ● org.jboss.vfs 	.

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
Security	<ul style="list-style-type: none"> ● org.picketbox ● org.jboss.as.security ● javax.security.jacc.api ● javax.security.auth.message.api 		
ServiceActivator		<ul style="list-style-type: none"> ● org.jboss.msc 	
Transactions	<ul style="list-style-type: none"> ● javax.transaction.api 	<ul style="list-style-type: none"> ● org.jboss.xts ● org.jboss.jts ● org.jboss.narayana.compensations 	
Undertow	<ul style="list-style-type: none"> ● javax.servlet.jstl.api ● javax.servlet.api ● javax.servlet.jsp.api ● javax.websocket.api 	<ul style="list-style-type: none"> ● io.undertow.core ● io.undertow.servlet ● io.undertow.jsp ● io.undertow.websocket ● io.undertow.js ● org.wildfly.clustering.web.api 	
Web Services	<ul style="list-style-type: none"> ● javax.jws.api ● javax.xml.soap.api ● javax.xml.ws.api 	<ul style="list-style-type: none"> ● org.jboss.ws.api ● org.jboss.ws.spi 	アプリケーションクライアントタイプでない場合は、条件付き依存関係が追加されます。

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
Weld (CDI)	<ul style="list-style-type: none"> ● javax.enterprise.api ● javax.inject.api 	<ul style="list-style-type: none"> ● javax.persistence.api ● org.javassist ● org.jboss.as.weld ● org.jboss.weld.core ● org.jboss.weld.probe ● org.jboss.weld.api ● org.jboss.weld.spi ● org.hibernate.validator.cdi 	デプロイメントに beans.xml ファイルが存在すること。

3.9.2. 含まれるモジュール

含まれるモジュールの完全なリストとこれらのモジュールがサポートされているかは、Red Hat カスタマーポータル[の「JBoss Enterprise Application Platform \(EAP\) 7 に含まれるモジュール」](#)を参照してください。

第4章 LOGGING

4.1. ロギング

ロギングとはアクティビティの記録 (ログ) を提供するアプリケーションからのメッセージを記録することです。

ログメッセージは、アプリケーションをデバッグする開発者や実稼働環境のアプリケーションを維持するシステム管理者に対して重要な情報を提供します。

ほとんどの最新の Java のロギングフレームワークには、正確な時間やメッセージの発信元などの詳細も含まれます。

4.1.1. サポート対象のアプリケーションロギングフレームワーク

JBoss LogManager は次のロギングフレームワークをサポートします。

- JBoss Logging (JBoss EAP に含まれる)
- [Apache Commons Logging](#)
- [Simple Logging Facade for Java \(SLF4J\)](#)
- [Apache log4j](#)
- [Java SE Logging \(java.util.logging\)](#)

JBoss LogManager では以下の API がサポートされます。

- JBoss Logging
- commons-logging
- SLF4J
- Log4j
- java.util.logging

JBoss LogManager では以下の SPI もサポートされます。

- java.util.logging Handler
- Log4j Appender



注記

Log4j API と **Log4J Appender** を使用している場合、オブジェクトは渡される前に **string** に変換されます。

4.2. JBOSS LOGGING FRAMEWORK を用いたロギング

4.2.1. JBoss Logging について

JBoss Logging は、JBoss EAP に含まれるアプリケーションロギングフレームワークです。JBoss Logging を使用すると、簡単にロギングをアプリケーションに追加できます。また、フレームワークを使用するアプリケーションにコードを追加し、定義された形式でログメッセージを送信できます。アプリケーションサーバーにアプリケーションがデプロイされると、これらのメッセージをサーバーでキャプチャーしたり、サーバーの設定に基づいて表示したり、ファイルに書き込んだりできます。

JBoss Logging では次の機能が提供されます。

- 革新的で使いやすい型指定されたロガー。型指定されたロガーは `org.jboss.logging.annotations.MessageLogger` でアノテーションが付けられたロガーインターフェースです。例は、[国際化されたロガー](#)、[メッセージ](#)、[例外の作成](#) を参照してください。
- 国際化およびローカリゼーションの完全なサポート。翻訳者は properties ファイルのメッセージバンドルを、開発者はインターフェースやアノテーションを使い作業を行います。詳細は、[「国際化と現地語化」](#) を参照してください。
- 実稼働用の型指定されたロガーを生成し、開発用の型指定されたロガーを実行時に生成する構築時ツール。

4.2.2. JBoss Logging を使用したアプリケーションへのロギングの追加

この手順では、JBoss Logging を使用してアプリケーションにロギングを追加する方法を示します。



重要

Maven を使用してプロジェクトをビルドする場合は、JBoss EAP Maven リポジトリを使用するよう Maven を設定する必要があります。詳細は、[「JBoss EAP Maven リポジトリの設定」](#) を参照してください。

1. JBoss Logging JAR ファイルがアプリケーションのビルドパスに指定されている必要があります。
 - Red Hat CodeReady Studio を使用して構築する場合は、**Project** メニューから **Properties** を選択し、**Targeted Runtimes** を選択して JBoss EAP のランタイムにチェックマークが付いていることを確認します。
 - Maven を使用してプロジェクトをビルドする場合は、JBoss Logging フレームワークにアクセスするために必ず **jboss-logging** 依存関係をプロジェクトの **pom.xml** ファイルに追加してください。

```
<dependency>
  <groupId>org.jboss.logging</groupId>
  <artifactId>jboss-logging</artifactId>
  <version>3.3.0.Final-redhat-1</version>
  <scope>provided</scope>
</dependency>
```

jboss-eap-javaee8 BOM は **jboss-logging** のバージョンを管理します。詳細は、[「プロジェクト依存関係の管理」](#) を参照してください。アプリケーションでのロギングの実例は、JBoss EAP に同梱される **logging** クイックスタートを参照してください。

JAR は、JBoss EAP がデプロイされたアプリケーションに提供するため、ビルドされたアプリケーションに含める必要はありません。

2. ロギングを追加する各クラスに対して、以下の手順を実行します。
 - a. 使用する JBoss Logging クラスネームスペースに対して `import` ステートメントを追加します。少なくとも、以下の `import` ステートメントが必要です。

```
import org.jboss.logging.Logger;
```

- b. **org.jboss.logging.Logger** のインスタンスを作成し、静的メソッド **Logger.getLogger(Class)** を呼び出して初期化します。各クラスに対してこれを単一のインスタンス変数として作成することが推奨されます。

```
private static final Logger LOGGER = Logger.getLogger>HelloWorld.class);
```

3. ログメッセージを送信するコードの **Logger** オブジェクトメソッドを呼び出します。**Logger** には、異なるタイプのメッセージに対して異なるパラメーターを持つさまざまなメソッドがあります。以下のメソッドを使用して対応するログレベルのログメッセージと **message** パラメーターを文字列として送信します。

```
LOGGER.debug("This is a debugging message.");
LOGGER.info("This is an informational message.");
LOGGER.error("Configuration file not found.");
LOGGER.trace("This is a trace message.");
LOGGER.fatal("A fatal error occurred.");
```

JBoss Logging メソッドの完全リストは、[Logging API](#) のドキュメンテーションを参照してください。

次の例では、プロパティファイルからアプリケーションのカスタマイズされた設定がロードされます。指定されたファイルが見つからない場合は、**ERROR** レベルログメッセージが記録されます。

例: JBoss Logging を用いたアプリケーションのロギング

```
import org.jboss.logging.Logger;
public class LocalSystemConfig
{
    private static final Logger LOGGER = Logger.getLogger(LocalSystemConfig.class);

    public Properties openCustomProperties(String configname) throws
    CustomConfigFileNotFoundException
    {
        Properties props = new Properties();
        try
        {
            LOGGER.info("Loading custom configuration from "+configname);
            props.load(new FileInputStream(configname));
        }
        catch(IOException e) //catch exception in case properties file does not exist
        {
            LOGGER.error("Custom configuration file ("+configname+) not found. Using defaults.");
            throw new CustomConfigFileNotFoundException(configname);
        }

        return props;
    }
}
```

4.3. デプロイメントごとのロギング

デプロイメントごとのロギングを使用すると、開発者はアプリケーションのロギング設定を事前に設定できます。アプリケーションがデプロイされると、定義された設定に従ってロギングが開始されます。この設定によって作成されたログファイルにはアプリケーションの動作に関する情報のみが含まれます。



注記

デプロイメントごとのロギング設定が行われない場合、すべてのアプリケーションとサーバーには **logging** サブシステムの設定が使用されます。

この方法では、システム全体のロギングを使用する利点と欠点があります。利点は、JBoss EAP インスタンスの管理者がサーバーロギング以外のロギングを設定する必要がないことです。欠点は、デプロイメントごとのロギング設定はサーバーの起動時に読み取り専用であるため、実行時に変更できないことです。

4.3.1. デプロイメントごとのロギングをアプリケーションに追加

アプリケーションのデプロイメントごとのロギングを設定するには、**logging.properties** 設定ファイルをデプロイメントに追加します。この設定ファイルは、JBoss Log Manager が基礎となるログマネージャーである場合にどのロギングファサードとも使用できるため、推奨されます。

設定ファイルが追加されるディレクトリーは、デプロイメント方法によって異なります。

- EAR デプロイメントの場合は、ロギング設定ファイルを **META-INF/** ディレクトリーにコピーします。
- WAR または JAR デプロイメントの場合は、ロギング設定ファイルを **WEB-INF/classes/** ディレクトリーにコピーします。



注記

Simple Logging Facade for Java (SLF4J) または **Apache log4j** を使用している場合は、**logging.properties** 設定ファイルが適しています。Apache log4j アペンダーを使用している場合は、**log4j.properties** 設定ファイルが必要になります。**jboss-logging.properties** 設定ファイルはレガシーデプロイメントのみでサポートされます。

logging.properties の設定

logging.properties ファイルはサーバーが起動し、**logging** サブシステムが起動するまで使用されます。**logging** サブシステムが設定に含まれない場合、サーバーはこのファイルの設定をサーバー全体のロギング設定として使用します。

JBoss ログマネージャーの設定オプション

ロガーオプション

- **loggers=<category>[,<category>,...]** - 設定するロガーカテゴリーのコンマ区切りのリストを指定します。ここで記載されていないカテゴリーは、以下のプロパティーから設定されません。
- **logger.<category>.level=<level>** - カテゴリーのレベルを指定します。このレベルは有効なレベルのいずれかになります。指定されない場合は、最も近い親のレベルが継承されます。

- **logger.<category>.handlers=<handler>[,<handler>,...]** - このログに割り当てるハンドラー名のコンマ区切りのリストを指定します。ハンドラーは同じプロパティファイルで設定する必要があります。
- **logger.<category>.filter=<filter>** - カテゴリーのフィルターを指定します。
- **logger.<category>.useParentHandlers=(true|false)** - ログメッセージを親ハンドラーにカスケードするかどうかを指定します。デフォルト値は **true** です。

ハンドラーオプション

- **handler.<name>=<className>** - インスタンス化するハンドラーのクラス名を指定します。このオプションは必須です。

注記

表4.1 可能なクラス名

名前	関連するクラス
Console	org.jboss.logmanager.handlers.ConsoleHandler
File	org.jboss.logmanager.handlers.FileHandler
Periodic	org.jboss.logmanager.handlers.PeriodicRotatingFileHandler
Size	org.jboss.logmanager.handlers.SizeRotatingFileHandler
Periodic Size	org.jboss.logmanager.handlers.PeriodicSizeRotatingFileHandler
Syslog	org.jboss.logmanager.handlers.SyslogHandler
Async	org.jboss.logmanager.handlers.AsyncHandler

Custom ハンドラーは関連するあらゆるクラスまたはモジュールを持つことができます。このハンドラーは **logging** サブシステムにあり、ユーザーは独自のログハンドラーを定義できます。

詳細は、JBoss EAP 『[設定ガイド](#)』の「[ログハンドラー](#)」を参照してください。

- **handler.<name>.level=<level>** - このハンドラーのレベルを制限します。指定されない場合は、ALL のデフォルト値が保持されます。

- **handler.<name>.encoding=<encoding>** - 文字エンコーディングを指定します (このハンドラタイプによりサポートされている場合)。指定されない場合は、ハンドラ固有のデフォルト値が使用されます。
- **handler.<name>.errorManager=<name>** - 使用するエラーマネージャーの名前を指定します。エラーマネージャーは同じプロパティファイルで設定する必要があります。指定されない場合は、エラーマネージャーが設定されません。
- **handler.<name>.filter=<name>** - カテゴリのフィルターを指定します。フィルターの定義の詳細は、フィルター式を参照してください。
- **handler.<name>.formatter=<name>** - 使用するフォーマッターの名前を指定します (このハンドラタイプによりサポートされている場合)。フォーマッターは同じプロパティファイルで設定する必要があります。指定されない場合、ほとんどのハンドラタイプのメッセージはログに記録されません。
- **handler.<name>.properties=<property>[,<property>,...]** - 追加的に設定する JavaBean 形式のプロパティを指定します。指定のプロパティが適切に変換されるように、基本的なタイプイントロスペクションが行われます。
JBoss Log Manager のすべてのファイルハンドラには、**fileName** の前に **append** を設定する必要があります。**handler.<name>.properties** でプロパティを指定する順番は、プロパティが設定される順番になります。
- **handler.<name>.constructorProperties=<property>[,<property>,...]** - 構築パラメーターとして使用する必要があるプロパティのリストを指定します。指定のプロパティが適切に変換されるように、基本的なタイプイントロスペクションが行われます。
- **handler.<name>.<property>=<value>** - 名前付きプロパティの値を設定します。
- **handler.<name>.module=<name>** - ハンドラが存在するモジュールの名前を指定します。

詳細は、JBoss EAP 『[設定ガイド](#)』の「[ログハンドラの属性](#)」を参照してください。

エラーマネージャーオプション

- **errorManager.<name>=<className>** - インスタンス化するエラーマネージャーのクラス名を指定します。このオプションは必須です。
- **errorManager.<name>.properties=<property>[,<property>,...]** - 追加的に設定する JavaBean 形式のプロパティを指定します。指定のプロパティが適切に変換されるように、基本的なタイプイントロスペクションが行われます。
- **errorManager.<name>.<property>=<value>** - 名前のついたプロパティの値を設定します。

フォーマッターオプション

- **formatter.<name>=<className>** - インスタンス化するフォーマッターのクラス名を指定します。このオプションは必須です。
- **formatter.<name>.properties=<property>[,<property>,...]** - 追加的に設定する JavaBean 形式のプロパティを指定します。指定のプロパティが適切に変換されるように、基本的なタイプイントロスペクションが行われます。
- **formatter.<name>.constructorProperties=<property>[,<property>,...]** - 構築パラメーターとして使用する必要があるプロパティのリストを指定します。指定のプロパティが適切に変換されるように、基本的なタイプイントロスペクションが行われます。

- `formatter.<name>.<property>=<value>` - 名前付きプロパティの値を設定します。

以下の例は、コンソールにログ記録する `logging.properties` ファイルの最低限の設定を示しています。

例: 最低限の `logging.properties` 設定

```
# Additional logger names to configure (root logger is always configured)
# loggers=

# Root logger level
logger.level=INFO

# Root logger handlers
logger.handlers=CONSOLE

# Console handler configuration
handler.CONSOLE=org.jboss.logmanager.handlers.ConsoleHandler
handler.CONSOLE.properties=autoFlush
handler.CONSOLE.autoFlush=true
handler.CONSOLE.formatter=PATTERN

# Formatter pattern configuration
formatter.PATTERN=org.jboss.logmanager.formatters.PatternFormatter
formatter.PATTERN.properties=pattern
formatter.PATTERN.pattern=%K{level}%d{HH:mm:ss,SSS} %-5p %C.%M(%L) [%c] %s%n
```

4.4. ロギングプロファイル

ロギングプロファイルは、デプロイされたアプリケーションに割り当てることができる独立したロギング設定のセットです。通常の `logging` サブシステム同様にロギングプロファイルはハンドラー、カテゴリ、およびルートロガーを定義できますが、他のプロファイルや主要な `logging` サブシステムを参照できません。設定が容易である点でロギングプロファイルは `logging` サブシステムと似ています。

ロギングプロファイルを使用すると、管理者は他のロギング設定に影響を与えずに1つ以上のアプリケーションに固有するロギング設定を作成することができます。各プロファイルはサーバー設定で定義されるため、ロギング設定を変更しても影響を受けるアプリケーションを再デプロイする必要はありません。

詳細は、JBoss EAP『[設定ガイド](#)』の「[ロギングプロファイルの設定](#)」を参照してください。

各ロギングプロファイルには以下の項目を設定できます。

- 一意な名前。この値は必須です。
- 任意の数のログハンドラー。
- 任意の数のログカテゴリ。
- 最大1つのルートロガー。

アプリケーションでは `Logging-Profile` 属性を使用して、`MANIFEST.MF` ファイルで使用するロギングプロファイルを指定できます。

4.4.1. アプリケーションでのロギングプロファイルの指定

アプリケーションでは、使用するロギングプロファイルを **MANIFEST.MF** ファイルで指定できます。



注記

このアプリケーションが使用するサーバー上に設定されたロギングプロファイルの名前を知っている必要があります。

ロギングプロファイル設定をアプリケーションに追加するには、**MANIFEST.MF** ファイルを編集します。

- アプリケーションに **MANIFEST.MF** ファイルがない場合は、ロギングプロファイル名を指定する以下の内容が含まれるファイルを作成します。

```
Manifest-Version: 1.0
Logging-Profile: LOGGING_PROFILE_NAME
```

- アプリケーションに **MANIFEST.MF** ファイルがすでにある場合は、ロギングプロファイル名を指定する以下の行を追加します。

```
Logging-Profile: LOGGING_PROFILE_NAME
```



注記

Maven および **maven-war-plugin** を使用している場合は、**MANIFEST.MF** ファイルを **src/main/resources/META-INF/** に置き、次の設定を **pom.xml** ファイルに追加します。

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <archive>
      <manifestFile>src/main/resources/META-INF/MANIFEST.MF</manifestFile>
    </archive>
  </configuration>
</plugin>
```

アプリケーションがデプロイされると、ログメッセージに対して指定されたロギングプロファイルの設定が使用されます。

ロギングプロファイルとそれを使用するアプリケーションの設定方法の例は、JBoss EAP『[設定ガイド](#)』の「[ロギングプロファイル設定の例](#)」を参照してください。

4.5. 国際化と現地語化

4.5.1. はじめに

4.5.1.1. 国際化

国際化とは、技術的な変更を行わずに異なる言語や地域に対してソフトウェアを適合させるソフトウェア設計のプロセスのことです。

4.5.1.2. 多言語化

多言語化とは、特定の地域や言語に対してロケール固有のコンポーネントやテキストの翻訳を追加することで、国際化されたソフトウェアを適合させるプロセスのことです。

4.5.2. JBoss Logging Tools の国際化および現地語化

JBoss Logging Tools は、ログメッセージ、例外メッセージ、および汎用文字列の国際化や現地語化のサポートを提供する Java API です。JBoss Logging Tools は翻訳のメカニズムを提供するだけでなく、各ログメッセージに対して一意な識別子のサポートも提供します。

国際化されたメッセージと例外は、**org.jboss.logging.annotations** アノテーションが付けられたインターフェース内でメソッド定義として作成されます。インターフェースを実装する必要はありません。JBoss Logging Tools がコンパイル時にインターフェースを実装します。定義すると、これらのメソッドを使用してコードでメッセージをログに記録したり、例外オブジェクトを取得したりできます。

JBoss Logging Tools によって作成される国際化されたロギングインターフェースや例外インターフェースは、特定の言語や地域に対する翻訳が含まれる各バンドルのプロパティファイルを作成して現地語化されます。JBoss Logging Tools は、トランスレーターが編集できる各バンドルに対してテンプレートプロパティファイルを生成できます。

JBoss Logging Tools は、プロジェクトの対象翻訳プロパティファイルごとに各バンドルの実装を作成します。必要なのはバンドルに定義されているメソッドを使用することのみで、JBoss Logging Tools は現在の地域設定に対して正しい実装が呼び出されるようにします。

メッセージ ID とプロジェクトコードは各ログメッセージの前に付けられる一意の識別子です。この一意の識別子をドキュメントで使用すると、ログメッセージの情報を簡単に検索することができます。適切なドキュメントでは、メッセージが書かれた言語に関係なく、ログメッセージの意味を識別子から判断できます。

JBoss Logging Tools には次の機能のサポートが含まれます。

MessageLogger

org.jboss.logging.annotations パッケージ内のこのインターフェースは、国際化されたログメッセージを定義するために使用されます。メッセージロガーインターフェースは **@MessageLogger** アノテーションが付けられます。

MessageBundle

このインターフェースは、翻訳可能な汎用メッセージと国際化されたメッセージが含まれる例外オブジェクトを定義するために使用できます。メッセージバンドルは、ログメッセージの作成には使用されません。メッセージバンドルインターフェースは、**@MessageBundle** アノテーションが付けられます。

国際化されたログメッセージ

これらのログメッセージは、**MessageLogger** のメソッドを定義して作成されます。メソッドは **@LogMessage** アノテーションと **@Message** アノテーションを付け、**@Message** の値属性を使用してログメッセージを指定する必要があります。国際化されたログメッセージはプロパティファイルで翻訳を提供することによりローカライズされます。

JBoss Logging Tools はコンパイル時に各翻訳に必要なロギングクラスを生成し、ランタイム時に現ロケールに対して適切なメソッドを呼び出します。

国際化された例外

国際化された例外は、MessageBundle で定義されたメソッドから返された例外オブジェクトです。これらのメッセージバンドルは、デフォルトの例外メッセージを定義するためにアノテーションを付けることができます。デフォルトのメッセージは、現在のロケールと一致するプロパティファイルに翻訳がある場合にその翻訳に置き換えられます。国際化された例外にも、プロジェクトコードとメッセージ ID を割り当てることができます。

国際化されたメッセージ

国際化されたメッセージは、**MessageBundle** で定義されたメソッドから返された文字列です。Java String オブジェクトを返すメッセージバンドルメソッドは、その文字列のデフォルトの内容 (メッセージと呼ばれます) を定義するためにアノテーションを付けることができます。デフォルトのメッセージは、現在のロケールと一致するプロパティファイルに翻訳がある場合にその翻訳に置き換えられます。

翻訳プロパティファイル

翻訳プロパティファイルは、1つのロケール、国、バリエーションに対する1つのインターフェースのメッセージの翻訳が含まれる Java プロパティファイルです。翻訳プロパティファイルは、メッセージを返すクラスを生成するために JBoss Logging Tools によって使用されます。

JBoss Logging Tools のプロジェクトコード

プロジェクトコードはメッセージのグループを識別する文字列です。プロジェクトコードは各ログメッセージの最初に表示され、メッセージ ID の前に付けられます。プロジェクトコードは **@MessageLogger** アノテーションの `projectCode` 属性で定義されます。



注記

新しいログメッセージプロジェクトコード接頭辞の完全なリストは、JBoss EAP 7.2 で使用されている [プロジェクトコード](#) を参照してください。

JBoss Logging Tools のメッセージ ID

メッセージ ID はプロジェクトコードと組み合わせてログメッセージを一意に識別する数字です。メッセージ ID は各ログメッセージの最初に表示され、メッセージのプロジェクトコードの後に付けられます。メッセージ ID は **@Message** アノテーションの `ID` 属性で定義されます。

JBoss EAP に同梱される **logging-tools** クイックスタートは、JBoss Logging Tools の多くの機能の例を提供する単純な Maven プロジェクトです。以降のコード例は、**logging-tools** クイックスタートから取得されました。

4.5.3. 国際化されたロガー、メッセージ、例外の作成

4.5.3.1. 国際化されたログメッセージの作成

JBoss Logging Tools を使用して **MessageLogger** インターフェースを作成することにより、国際化されたログメッセージを作成できます。



注記

ここでは、ログメッセージのすべてのオプション機能または国際化について説明しません。

1. JBoss EAP Maven レポジトリを使用するよう Maven を設定します (まだそのように設定していない場合)。詳細は、「[Maven 設定を使用した JBoss EAP Maven リポジトリの設定](#)」を参照してください。
2. JBoss Logging Tools を使用するようプロジェクトの **pom.xml** ファイルを設定します。詳細は、「[JBoss Logging Tools の Maven 設定](#)」を参照してください。
3. ログメッセージ定義を含めるために Java インターフェースをプロジェクトに追加して、メッセージロガーインターフェースを作成します。

定義するログメッセージの内容がわかるようインターフェースに名前を付けます。ログメッセージインターフェースの要件は次のとおりです。

- **@org.jboss.logging.annotations.MessageLogger** アノテーションを付ける必要があります。
- オプションで、**org.jboss.logging.BasicLogger** を拡張できます。
- インターフェースと同じ型のメッセージロガーであるフィールドをインターフェースで定義する必要があります。これは、**@org.jboss.logging.Logger** の **getMessageLogger()** を使用して行います。

例: メッセージロガーの作成

```
package com.company.accounts.loggers;

import org.jboss.logging.BasicLogger;
import org.jboss.logging.Logger;
import org.jboss.logging.annotations.MessageLogger;

@MessageLogger(projectCode="")
interface AccountsLogger extends BasicLogger {
    AccountsLogger LOGGER = Logger.getMessageLogger(
        AccountsLogger.class,
        AccountsLogger.class.getPackage().getName() );
}
```

4. 各ログメッセージのインターフェースにメソッド定義を追加します。ログメッセージの各メソッドにその内容を表す名前を付けます。各メソッドの要件は次のとおりです。

- メソッドは **void** を返す必要があります。
- **@org.jboss.logging.annotation.LogMessage** アノテーションを付ける必要があります。
- **@org.jboss.logging.annotations.Message** アノテーションを付ける必要があります。
- デフォルトのログレベルは **INFO** です。
- **@org.jboss.logging.annotations.Message** の値属性にはデフォルトのログインメッセージが含まれます。このメッセージは翻訳がない場合に使用されます。

```
@LogMessage
@Message(value = "Customer query failed, Database not available.")
void customerQueryFailDBClosed();
```

5. メッセージをログに記録する必要があるコードで呼び出しをインターフェースメソッドに追加してメソッドを呼び出します。インターフェースの実装を作成する必要はありません。これは、プロジェクトがコンパイルされる時にアノテーションプロセッサにより行われます。

```
AccountsLogger.LOGGER.customerQueryFailDBClosed();
```

カスタムのロガーは **BasicLogger** からサフクラス化されるため、**BasicLogger** のロギングメソッドを使用することもできます。国際化されていないメッセージをログに記録するために他のロガーを作成する必要はありません。

```
AccountsLogger.LOGGER.error("Invalid query syntax.");
```

- プロジェクトで、現地語化できる1つ以上の国際化されたロガーがサポートされるようになります。



注記

JBoss EAP に同梱される **logging-tools** クイックスタートは、JBoss Logging Tools の使用例を提供する単純な Maven プロジェクトです。

4.5.3.2. 国際化されたメッセージの作成と使用

この手順では、国際化された例外を作成および使用方法を示します。



注記

本項では、これらのメッセージの現地語化に関するすべてのオプション機能またはプロセスについて説明しません。

- JBoss EAP Maven レポジトリを使用するよう Maven を設定します (まだそのように設定していない場合)。詳細は、[「Maven 設定を使用した JBoss EAP Maven レポジトリの設定」](#) を参照してください。
- JBoss Logging Tools を使用するようプロジェクトの **pom.xml** ファイルを設定します。詳細は、[「JBoss Logging Tools の Maven 設定」](#) を参照してください。
- 例外のインターフェースを作成します。JBoss Logging Tools はインターフェースで国際化されたメッセージを定義します。含まれるメッセージのインターフェースにその内容を表す名前を付けます。インターフェースの要件は以下のとおりです。
 - public** として宣言する必要があります。
 - @org.jboss.logging.annotations.MessageBundle** アノテーションを付ける必要があります。
 - インターフェースと同じ型のメッセージバンドルであるフィールドをインターフェースが定義する必要があります。

例: MessageBundle インターフェースの作成

```
@MessageBundle(projectCode="")
public interface GreetingMessageBundle {
    GreetingMessageBundle MESSAGES =
        Messages.getBundle(GreetingMessageBundle.class);
}
```



注記

`Messages.getBundle(GreetingMessagesBundle.class)` を呼び出すのは `Messages.getBundle(GreetingMessagesBundle.class, Locale.getDefault())` を呼び出すのと同様です。

`Locale.getDefault()` は、Java Virtual Machine のこのインスタンスのデフォルトロケールに対する現在の値を取得します。起動時に、ホストの環境に基づいて Java Virtual Machine によりデフォルトのロケールが設定されます。これは、ロケールが明示的に指定されていない場合に、ロケールに関連する多くのメソッドによって使用されます。これは、`setDefault` メソッドで変更できます。

詳細は、JBoss EAP 『[設定ガイド](#)』の「[サーバーのデフォルトロケールの設定](#)」を参照してください。

4. 各メッセージのインターフェースにメソッド定義を追加します。メッセージに対する各メソッドにその内容を表す名前を付けます。各メソッドの要件は次のとおりです。
 - 型 `String` のオブジェクトを返す必要があります。
 - `@org.jboss.logging.annotations.Message` アノテーションを付ける必要があります。
 - デフォルトメッセージに `@org.jboss.logging.annotations.Message` の値属性を設定する必要があります。このメッセージは翻訳がない場合に使用されます。

```
@Message(value = "Hello world.")
String helloworldString();
```

5. メッセージを取得する必要があるアプリケーションでインターフェースメソッドを呼び出します。

```
System.out.println(helloworldString());
```

プロジェクトで、現地語化できる国際化されたメッセージ文字列がサポートされるようになります。



注記

使用できる完全な例は、JBoss EAP に同梱される `logging-tools` クイックスタートを参照してください。

4.5.3.3. 国際化された例外の作成

JBoss Logging Tools を使用して、国際化された例外を作成および使用できます。

以下の手順では、Red Hat CodeReady Studio または Maven のいずれかを使用してビルドされた既存のソフトウェアプロジェクトに、国際化された例外を追加することを前提としています。



注記

ここでは、これらの例外のすべてのオプション機能または国際化のプロセスについて説明しません。

1. JBoss Logging Tools を使用するようプロジェクトの **pom.xml** ファイルを設定します。詳細は、「[JBoss Logging Tools の Maven 設定](#)」を参照してください。
2. 例外のインターフェースを作成します。JBoss Logging Tools はインターフェースで国際化されたメッセージを定義します。各インターフェースに定義する例外を表す名前を付けます。インターフェースの要件は以下のとおりです。
 - **public** として宣言する必要があります。
 - **@MessageBundle** アノテーションを付ける必要があります。
 - インターフェースと同じ型のメッセージバンドルであるフィールドをインターフェースが定義する必要があります。

例: **ExceptionBundle** インターフェースの作成

```
@MessageBundle(projectCode="")
public interface ExceptionBundle {
    ExceptionBundle EXCEPTIONS = Messages.getBundle(ExceptionBundle.class);
}
```

3. 各例外のインターフェースにメソッド定義を追加します。例外に対する各メソッドにその内容を表す名前を付けます。各メソッドの要件は次のとおりです。
 - **Exception** オブジェクトまたは **Exception** のサブタイプを返す必要があります。
 - **@org.jboss.logging.annotations.Message** アノテーションを付ける必要があります。
 - デフォルトの例外メッセージに **@org.jboss.logging.annotations.Message** の値属性を設定する必要があります。このメッセージは翻訳がない場合に使用されます。
 - メッセージ文字列の他にパラメーターを必要とするコンストラクターが返される例外にある場合は、**@Param** アノテーションを使用してこれらのパラメーターをメソッド定義に提供する必要があります。パラメーターは、例外のコンストラクターと同じ型および順番である必要があります。

```
@Message(value = "The config file could not be opened.")
IOException configFileAccessError();

@Message(id = 13230, value = "Date string '%s' was invalid.")
ParseException dateWasInvalid(String dateString, @Param int errorOffset);
```

4. 例外を取得する必要があるコードでインターフェースメソッドを呼び出します。メソッドによって例外は発生されず、発生できるな例外オブジェクトがメソッドによって返されます。

```
try {
    propsInFile=new File(configname);
    props.load(new FileInputStream(propsInFile));
}
catch(IOException ioex) {
    //in case props file does not exist
    throw ExceptionBundle.EXCEPTIONS.configFileAccessError();
}
```

プロジェクトで、現地語化できる国際化された例外がサポートされるようになります。



注記

使用できる完全な例は、JBoss EAP に同梱される **logging-tools** クイックスタートを参照してください。

4.5.4. 国際化されたロガー、メッセージ、例外の現地語化

4.5.4.1. Maven での新しい翻訳プロパティーファイルの作成

Maven で構築されたプロジェクトでは、含まれる **MessageLogger** と **MessageBundle** それぞれに対して空の翻訳プロパティーファイルを生成できます。これらのファイルは新しい翻訳プロパティーファイルとして使用することができます。

新しい翻訳プロパティーファイルを生成するよう Maven プロジェクトを設定する手順は次のとおりです。

要件

- 作業用の Maven プロジェクトがすでに存在している必要があります。
- JBoss Logging Tools に対してプロジェクトが設定されていなければなりません。
- 国際化されたログメッセージや例外を定義する1つ以上のインターフェースがプロジェクトに含まれていなければなりません。

翻訳プロパティーファイルの生成

1. **-AgeneratedTranslationFilePath** コンパイラ引数を Maven コンパイラプラグイン設定に追加し、新しいファイルが作成されるパスを割り当てます。
この設定では、Maven プロジェクトの **target/generated-translation-files** ディレクトリーに新しいファイルが作成されます。

例: 翻訳ファイルパスの定義

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>2.3.2</version>
  <configuration>
    <source>1.6</source>
    <target>1.6</target>
    <compilerArgument>
      -AgeneratedTranslationFilesPath=${project.basedir}/target/generated-translation-files
    </compilerArgument>
    <showDeprecation>>true</showDeprecation>
  </configuration>
</plugin>
```

2. Maven を使用してプロジェクトをビルドします。

```
$ mvn compile
```

@MessageBundle または **@MessageLogger** アノテーションが付けられた各インターフェースに対して1つのプロパティーファイルが作成されます。

- 各インターフェースが宣言された Java パッケージに対応するサブディレクトリーに新しいファイルが作成されます。
- 新しい各ファイルには、以下のパターンで名前が付けられます。ここで、**INTERFACE_NAME** はファイルを生成するために使用するインターフェースの名前です。

```
INTERFACE_NAME.i18n_locale_COUNTRY_VARIANT.properties
```

生成されたファイルは新しい翻訳の基礎としてプロジェクトにコピーできます。



注記

使用できる完全な例は、JBoss EAP に同梱される **logging-tools** クイックスタートを参照してください。

4.5.4.2. 国際化されたロガー、例外、またはメッセージの翻訳

プロパティファイルは、JBoss Logging Tools を使用してインターフェースで定義されたロギングおよび例外メッセージの翻訳を提供します。

次の手順は、翻訳プロパティファイルの作成方法と使用方法を示しています。この手順では、国際化された例外またはログメッセージに対して1つ以上のインターフェースがすでに定義されているプロジェクトが存在することを前提にしています。

要件

- 作業用の Maven プロジェクトがすでに存在している必要があります。
- JBoss Logging Tools に対してプロジェクトが設定されていなければなりません。
- 国際化されたログメッセージや例外を定義する1つ以上のインターフェースがプロジェクトに含まれていなければなりません。
- テンプレート翻訳プロパティファイルを生成するようプロジェクトが設定されている必要があります。

国際化されたロガー、例外、またはメッセージの翻訳

1. 以下のコマンドを実行して、テンプレート翻訳プロパティファイルを作成します。

```
$ mvn compile
```

2. 翻訳したいインターフェースのテンプレートを、テンプレートが作成されたディレクトリーからプロジェクトの **src/main/resources** ディレクトリーにコピーします。プロパティファイルは翻訳するインターフェースと同じパッケージに存在する必要があります。
3. コピーしたテンプレートファイルの名前を変更して、そのテンプレートに含まれる言語を指定します。例: GreeterLogger.i18n_fr_FR.properties
4. 新しい翻訳プロパティファイルの内容を編集し、適切な翻訳が含まれるようにします。

```
# Level: Logger.Level.INFO
# Message: Hello message sent.
logHelloMessageSent=Bonjour message envoy .
```

- 5. テンプレートをコピーし、バンドルの各翻訳のために変更するプロセスを繰り返します。

プロジェクトに1つ以上のメッセージバンドルまたはロガーバンドルに対する翻訳が含まれるようになります。プロジェクトをビルドすると、提供された翻訳が含まれるログメッセージに対して適切なクラスが生成されます。JBoss Logging Tools は、アプリケーションサーバーの現在のロケールに合わせて適切なクラスを自動的に使用するため、明示的にメソッドを呼び出したり、特定言語のパラメーターを提供したりする必要はありません。

生成されたクラスのソースコードは `target/generated-sources/annotations/` で確認できます。

4.5.5. 国際化されたログメッセージのカスタマイズ

4.5.5.1. ログメッセージへのメッセージ ID とプロジェクトコードの追加

この手順は、メッセージ ID とプロジェクトコードを JBoss Logging Tools を使用して作成された国際化済みログメッセージへ追加する方法を示しています。ログメッセージがログで表示されるようにするには、プロジェクトコードとメッセージ ID の両方が必要です。メッセージにプロジェクトコードとメッセージ ID の両方がない場合は、どちらも表示されません。

要件

1. 国際化されたログメッセージが含まれるプロジェクトが存在する必要があります。 [「国際化されたログメッセージの作成」](#) を参照してください。
2. 使用するプロジェクトコードを知っている必要があります。プロジェクトコードを1つ使用することも、各インターフェースに異なる複数のコードを定義することも可能です。

ログメッセージへのメッセージ ID とプロジェクトコードの追加

1. カスタムのロガーインターフェースに付けられる `@MessageLogger` アノテーションの `projectCode` 属性を使用してプロジェクトコードを指定します。インターフェースに定義されるすべてのメッセージがこのプロジェクトコードを使用します。

```
@MessageLogger(projectCode="ACCNTS")
interface AccountsLogger extends BasicLogger {

}
```

2. メッセージを定義するメソッドに付けられる `@Message` アノテーションの `id` 属性を使用して、各メッセージのメッセージ ID を指定します。

```
@LogMessage
@Message(id=43, value = "Customer query failed, Database not available.") void
customerQueryFailDBClosed();
```

3. メッセージ ID とプロジェクトコードの両方が関連付けられたログメッセージでは、メッセージ ID とプロジェクトコードがログに記録されたメッセージの前に付けられます。

```
10:55:50,638 INFO [com.company.accounts.ejb] (MSC service thread 1-4) ACCNTS000043:
Customer query failed, Database not available.
```

4.5.5.2. メッセージのログレベル設定

JBoss Logging Tools のインターフェースによって定義されるメッセージのデフォルトのログレベルは **INFO** です。ロギングメソッドに付けられた **@LogMessage** アノテーションの **level** 属性を用いて異なるログレベルを指定することが可能です。異なるログレベルを指定するには、以下の手順を実行します。

1. ログメッセージメソッド定義の **@LogMessage** アノテーションに **level** 属性を追加します。
2. **level** 属性を使用してこのメッセージにログレベルを割り当てます。 **level** の有効値は **org.jboss.logging.Logger.Level** で定義された **DEBUG**、**ERROR**、**FATAL**、**INFO**、**TRACE**、および **WARN** の 6 つの列挙定数です。

```
import org.jboss.logging.Logger.Level;

@LogMessage(level=Level.ERROR)
@Message(value = "Customer query failed, Database not available.")
void customerQueryFailDBClosed();
```

上記の例のロギングメソッドを呼び出すと、**ERROR** レベルのログメッセージが作成されます。

```
10:55:50,638 ERROR [com.company.app.Main] (MSC service thread 1-4)
Customer query failed, Database not available.
```

4.5.5.3. パラメーターによるログメッセージのカスタマイズ

カスタムのロギングメソッドはパラメーターを定義できます。これらのパラメーターを使用してログメッセージに表示される追加情報を渡すことが可能です。ログメッセージでパラメーターが表示される場所は、明示的なインデクシングか通常のインデクシングを使用してメッセージ自体に指定されます。

パラメーターによるログメッセージのカスタマイズ

1. すべての型のパラメーターをメソッド定義に追加します。型に関係なくパラメーターの String 表現がメッセージに表示されます。
2. ログメッセージにパラメーター参照を追加します。参照は明示的なインデックスまたは通常のインデックスを使用できます。
 - 通常のインデックスを使用するには、各パラメーターを表示したいメッセージ文字列に **%s** 文字を挿入します。**%s** の最初のインスタンスにより最初のパラメーターが挿入され、2 番目のインスタンスにより 2 番目のパラメーターが挿入されます。
 - 明示的なインデックスを使用するには、文字 **##\$s** をメッセージに挿入します。ここで、**#** は表示したいパラメーターの数を示します。

明示的なインデックスを使用すると、メッセージのパラメーター参照の順番がメソッドで定義される順番とは異なるようになります。これは、異なるパラメーターの順番が必要になる可能性がある翻訳済みメッセージで重要になります。



重要

指定されたメッセージでは、パラメーターの数とパラメーターへの参照の数と同じでなければなりません。同じでない場合、コードがコンパイルされません。**@Cause** アノテーションが付けられたパラメーターはパラメーターの数には含まれません。

以下に、通常のインデックスを使用したメッセージパラメーターの例を示します。

```
@LogMessage(level=Logger.Level.DEBUG)
@Message(id=2, value="Customer query failed, customerid:%s, user:%s")
void customerLookupFailed(Long customerid, String username);
```

以下に、明示的なインデックスを使用したメッセージパラメーターの例を示します。

```
@LogMessage(level=Logger.Level.DEBUG)
@Message(id=2, value="Customer query failed, user:%2$s, customerid:%1$s")
void customerLookupFailed(Long customerid, String username);
```

4.5.5.4. 例外をログメッセージの原因として指定

JBoss Logging Tools では、カスタムログインメソッドのパラメーターの1つをメッセージの原因として定義することができます。定義するには、このパラメーターを **Throwable** 型またはいずれかのサブクラスにし、**@Cause** アノテーションを付ける必要があります。このパラメーターは、他のパラメーターのようにログメッセージで参照することはできず、ログメッセージの後に表示されます。

次の手順は、**@Cause** パラメーターを使用して「原因となる」例外を示し、ロギングメソッドを更新する方法を表しています。この機能に追加したい国際化されたロギングメッセージがすでに作成されていることを前提とします。

例外をログメッセージの原因として指定

1. **Throwable** 型のパラメーターまたはサブクラスをメソッドに追加します。

```
@LogMessage
@Message(id=404, value="Loading configuration failed. Config file:%s")
void loadConfigFailed(Exception ex, File file);
```

2. パラメーターに **@Cause** アノテーションを追加します。

```
import org.jboss.logging.annotations.Cause

@LogMessage
@Message(value = "Loading configuration failed. Config file: %s")
void loadConfigFailed(@Cause Exception ex, File file);
```

3. メソッドを呼び出します。コードでメソッドが呼び出されると、正しい型のオブジェクトが渡され、ログメッセージの後に表示されます。

```
try
{
    confFile=new File(filename);
    props.load(new FileInputStream(confFile));
}
catch(Exception ex) //in case properties file cannot be read
{
    ConfigLogger.LOGGER.loadConfigFailed(ex, filename);
}
```

コードによって **FileNotFoundException** 型の例外が発生した場合、上記コード例の出力は次のようになります。

```
10:50:14,675 INFO [com.company.app.Main] (MSC service thread 1-3) Loading configuration failed.
Config file: customised.properties
java.io.FileNotFoundException: customised.properties (No such file or directory)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:120)
  at com.company.app.demo.Main.openCustomProperties(Main.java:70)
  at com.company.app.Main.go(Main.java:53)
  at com.company.app.Main.main(Main.java:43)
```

4.5.6. 国際化された例外のカスタマイズ

4.5.6.1. メッセージ ID およびプロジェクトコードの例外メッセージへの追加

メッセージ ID およびプロジェクトコードは、国際化された例外によって表示される各メッセージの前に付けられる一意の識別子です。これらの識別コードにより、アプリケーションですべての例外メッセージの参照を作成できます。これにより、習得していない言語で書かれた例外メッセージの意味を検索できます。

以下の手順は、JBoss Logging Tools を使用して作成された国際化済み例外メッセージにメッセージ ID とプロジェクトコードを追加する方法を示しています。

要件

1. 国際化された例外が含まれるプロジェクトが存在する必要があります。詳細は、「[国際化された例外の作成](#)」を参照してください。
2. 使用するプロジェクトコードを知っている必要があります。プロジェクトコードを1つ使用することも、各インターフェースに異なる複数のコードを定義することも可能です。

メッセージ ID およびプロジェクトコードの例外メッセージへの追加

1. 例外バンドルインターフェースに付けられる **@MessageBundle** アノテーションの **projectCode** 属性を使用して、プロジェクトコードを指定します。インターフェースに定義されるすべてのメッセージがこのプロジェクトコードを使用します。

```
@MessageBundle(projectCode="ACCTS")
interface ExceptionBundle
{
    ExceptionBundle EXCEPTIONS = Messages.getBundle(ExceptionBundle.class);
}
```

2. 例外を定義するメソッドに付けられる **@Message** アノテーションの **id** 属性を使用して、各例外に対してメッセージ ID を指定します。

```
@Message(id=143, value = "The config file could not be opened.")
IOException configFileAccessError();
```



重要

プロジェクトコードとメッセージ ID を両方持つメッセージでは、メッセージの前にプロジェクトコードとメッセージ ID が表示されます。プロジェクトコードとメッセージ ID の両方がない場合は、どちらも表示されません。

例: 国際化された例外

この例外バンドルインターフェースの例では、「ACCTS」のプロジェクトコードを使用しています。IDが「143」である例外メソッドが1つ含まれています。

```
@MessageBundle(projectCode="ACCTS")
interface ExceptionBundle
{
    ExceptionBundle EXCEPTIONS = Messages.getBundle(ExceptionBundle.class);

    @Message(id=143, value = "The config file could not be opened.")
    IOException configFileAccessError();
}
```

次のコードを使用すると、例外オブジェクトを取得および発生できます。

```
throw ExceptionBundle.EXCEPTIONS.configFileAccessError();
```

これにより、次のような例外メッセージが表示されます。

```
Exception in thread "main" java.io.IOException: ACCTS000143: The config file could not be opened.
at com.company.accounts.Main.openCustomProperties(Main.java:78)
at com.company.accounts.Main.go(Main.java:53)
at com.company.accounts.Main.main(Main.java:43)
```

4.5.6.2. パラメーターによる例外メッセージのカスタマイズ

例外を定義する例外バンドルメソッドでは、パラメーターを指定して例外メッセージに表示される追加情報を渡すことが可能です。例外メッセージでのパラメーターの正確な位置は、明示的なインデックスまたは通常のインデックスを使用してメッセージ自体に指定されます。

パラメーターによる例外メッセージのカスタマイズ

- すべての型のパラメーターをメソッド定義に追加します。型に関係なくパラメーターの String 表現がメッセージに表示されます。
- 例外メッセージにパラメーター参照を追加します。参照は明示的なインデックスまたは通常のインデックスを使用できます。
 - 通常のインデックスを使用するには、各パラメーターを表示したいメッセージ文字列に **%s** 文字を挿入します。**%s** の最初のインスタンスにより最初のパラメーターが挿入され、2 番目のインスタンスにより 2 番目のパラメーターが挿入されます。
 - 明示的なインデックスを使用するには、文字 **##\$s** をメッセージに挿入します。ここで、**#** は表示したいパラメーターの数を示します。

明示的なインデックスを使用すると、メッセージのパラメーター参照の順番がメソッドで定義される順番とは異なるようになります。これは、異なるパラメーターの順番が必要になる可能性がある翻訳済みメッセージで重要になります。



重要

指定されたメッセージでは、パラメーターの数とパラメーターへの参照の数が同じでなければなりません。**@Cause** アノテーションが付けられたパラメーターはパラメーターの数には含まれません。

例: 通常のインデックスの使用

```
@Message(id=2, value="Customer query failed, customerid:%s, user:%s")
void customerLookupFailed(Long customerid, String username);
```

例: 明示的なインデックスの使用

```
@Message(id=2, value="Customer query failed, user:%2$s, customerid:%1$s")
void customerLookupFailed(Long customerid, String username);
```

4.5.6.3. 別の例外の原因として1つの例外を指定

例外バンドルメソッドより返された例外に対し、他の例外を基礎となる原因として指定することができます。指定するには、パラメーターをメソッドに追加し、パラメーターに **@Cause** アノテーションを付けます。このパラメーターを使用して原因となる例外を渡します。このパラメーターを例外メッセージで参照することはできません。

次の手順は、**@Cause** パラメーターを使用して原因となる例外を示し、例外バンドルよりメソッドを更新する方法を表しています。この機能に追加したい国際化された例外バンドルがすでに作成されていることを前提とします。

1. **Throwable** 型のパラメーターまたはサブクラスをメソッドに追加します。

```
@Message(id=328, value = "Error calculating: %s.")
ArithmeticException calculationError(Throwable cause, String msg);
```

2. パラメーターに **@Cause** アノテーションを追加します。

```
import org.jboss.logging.annotations.Cause

@Message(id=328, value = "Error calculating: %s.")
ArithmeticException calculationError(@Cause Throwable cause, String msg);
```

3. 例外オブジェクトを取得するため、インターフェースメソッドを呼び出します。**catch** ブロックから新しい例外を発生させ、キャッチした例外を原因として指定するのが最も一般的なユースケースです。

```
try
{
    ...
}
catch(Exception ex)
{
    throw ExceptionBundle.EXCEPTIONS.calculationError(
        ex, "calculating payment due per day");
}
```

以下に、例外を別の例外の原因として指定する例を示します。この例外バンドルでは、**ArithmeticException** 型の例外を返す単一のメソッドを定義します。

```
@MessageBundle(projectCode = "TPS")
interface CalcExceptionBundle
{
```

```

CalcExceptionBundle EXCEPTIONS = Messages.getBundle(CalcExceptionBundle.class);

@Message(id=328, value = "Error calculating: %s.")
ArithmeticException calcError(@Cause Throwable cause, String value);
}

```

以下は、整数のゼロ除算が行われると例外が発生する操作の例になります。例外がキャッチされ、その最初の例外を原因として使用して新しい例外が作成されます。

```

int totalDue = 5;
int daysToPay = 0;
int amountPerDay;

try
{
    amountPerDay = totalDue/daysToPay;
}
catch (Exception ex)
{
    throw CalcExceptionBundle.EXCEPTIONS.calcError(ex, "payments per day");
}

```

以下は、前の例から生成された例外メッセージになります。

```

Exception in thread "main" java.lang.ArithmeticException: TPS000328: Error calculating: payments
per day.
    at com.company.accounts.Main.go(Main.java:58)
    at com.company.accounts.Main.main(Main.java:43)
Caused by: java.lang.ArithmeticException: / by zero
    at com.company.accounts.Main.go(Main.java:54)
    ... 1 more

```

4.5.7. JBoss Logging Tools のリファレンス

4.5.7.1. JBoss Logging Tools の Maven 設定

以下の手順では、国際化のために JBoss Logging と JBoss Logging Tools を使用するよう Maven プロジェクトを設定します。

1. JBoss EAP レポジトリを使用するよう Maven を設定します (まだそのように設定していない場合)。詳細は、「[Maven 設定を使用した JBoss EAP Maven リポジトリの設定](#)」を参照してください。

pom.xml ファイルの **<dependencyManagement>** セクションに **jboss-eap-javaee8** BOM を含めます。

```

<dependencyManagement>
  <dependencies>
    <!-- JBoss distributes a complete set of Java EE APIs including
         a Bill of Materials (BOM). A BOM specifies the versions of a "stack" (or
         a collection) of artifacts. We use this here so that we always get the correct versions of
         artifacts.
         Here we use the jboss-javaee-7.0 stack (you can
         read this as the JBoss stack of the Java EE APIs). You can actually
         use this stack with any version of JBoss EAP that implements Java EE. -->

```

```

<dependency>
  <groupId>org.jboss.bom</groupId>
  <artifactId>jboss-eap-javaee8</artifactId>
  <version>7.2.0.GA</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
<dependencies>
</dependencies>
</dependencyManagement>

```

2. Maven 依存関係をプロジェクトの **pom.xml** ファイルに追加します。

- a. JBoss Logging フレームワークにアクセスするために **jboss-logging** 依存関係を追加します。
- b. JBoss Logging Tools を使用する場合は、**jboss-logging-processor** 依存関係も追加します。これら両方の依存関係は、前の手順で追加された JBoss EAP BOM で利用できます。したがって、各依存関係のスコープ要素は示されているように **provided** に設定できます。

```

<!-- Add the JBoss Logging Tools dependencies -->
<!-- The jboss-logging API -->
<dependency>
  <groupId>org.jboss.logging</groupId>
  <artifactId>jboss-logging</artifactId>
  <scope>provided</scope>
</dependency>
<!-- Add the jboss-logging-tools processor if you are using JBoss Tools -->
<dependency>
  <groupId>org.jboss.logging</groupId>
  <artifactId>jboss-logging-processor</artifactId>
  <scope>provided</scope>
</dependency>

```

3. maven-compiler-plugin のバージョンは **3.1** 以上であり、**1.8** のターゲットソースおよび生成されたソースに対して設定する必要があります。

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
  </configuration>
</plugin>

```



注記

JBoss Logging Tools を使用するよう設定された **pom.xml** ファイルの完全な例は、JBoss EAP に同梱される **logging-tools** クイックスタートを参照してください。

4.5.7.2. 翻訳プロパティファイルの形式

JBoss Logging Tools でのメッセージの翻訳に使用されるプロパティファイルは標準的な Java プロパティファイルです。このファイルの形式は、[java.util.Properties](#) クラスドキュメンテーションに記載されている単純な行指向の **key=value** ペア形式です。

ファイル名の形式は次のようになります。

```
InterfaceName.i18n_locale_COUNTRY_VARIANT.properties
```

- **InterfaceName** は翻訳が適用されるインターフェースの名前です。
- **locale**、**COUNTRY**、および **VARIANT** は翻訳が適用される地域設定を識別します。
- **locale** と **COUNTRY** は ISO-639 および ISO-3166 言語および国コードを使用して言語と国を指定します。**COUNTRY** は任意です。
- **VARIANT** は特定のオペレーティングシステムまたはブラウザのみに適用される翻訳を識別するために使用できる任意の識別子です。

翻訳ファイルに含まれるプロパティは翻訳されるインターフェースのメソッドの名前です。プロパティに割り当てられた値が翻訳になります。メソッドがオーバーロードされる場合、これはドットとパラメーターの数を名前に付加することによって示されます。翻訳のメソッドは、異なる数のパラメーターを提供することによってのみオーバーロードできます。

例: 翻訳プロパティファイル

ファイル名: **GreeterService.i18n_fr_FR_POSIX.properties**

```
# Level: Logger.Level.INFO
# Message: Hello message sent.
logHelloMessageSent=Bonjour message envoyé.
```

4.5.7.3. JBoss Logging Tools のアノテーションに関するリファレンス

JBoss Logging では、ログメッセージや文字列、例外の国際化や現地語化に使用する以下のアノテーションが定義されています。

表4.2 JBoss Logging Tools のアノテーション

アノテーション	ターゲット	説明	属性
@MessageBundle	インターフェース	インターフェースをメッセージバンドルとして定義します。	projectCode
@MessageLogger	インターフェース	インターフェースをメッセージロガーとして定義します。	projectCode
@Message	方法	メッセージバンドルとメッセージロガーで使用できます。メッセージバンドルでは、現地語化された String または Exception オブジェクトを返すメソッドとして定義されます。メッセージロガーでは、現地語化されたロガーとしてメソッドが定義されます。	value, id

アノテーション	ターゲット	説明	属性
@LogMessage	方法	メッセージロガーのメソッドをロギングメソッドとして定義します。	level (デフォルトは INFO)
@Cause	パラメーター	ログメッセージまたは他の例外が発生したときに例外を渡すパラメーターとして定義します。	-
@Param	パラメーター	例外のコンストラクターへ渡されるパラメーターとして定義します。	-

4.5.7.4. JBoss EAP で使用されるプロジェクトコード

以下の表は、JBoss EAP 7.2 で使用されるすべてのプロジェクトコードとそのプロジェクトコードが属する Maven モジュールの一覧です。

表4.3 JBoss EAP で使用されるプロジェクトコード

Maven モジュール	プロジェクトコード
appclient	WFLYAC
batch/extension-jberet	WFLYBATCH
batch/extension	WFLYBATCH-DEPRECATED
batch/jberet	WFLYBAT
bean-validation	WFLYBV
controller-client	WFLYCC
controller	WFLYCTL
clustering/common	WFLYCLCOM
clustering/ejb/infinispan	WFLYCLEJBINF
clustering/infinispan/extension	WFLYCLINF
clustering/jgroups/extension	WFLYCLJG
clustering/server	WFLYCLSV
clustering/web/infinispan	WFLYCLWEBINF

Maven モジュール	プロジェクトコード
connector	WFLYJCA
deployment-repository	WFLYDR
deployment-scanner	WFLYDS
domain-http	WFLYDMHTTP
domain-management	WFLYDM
ee	WFLYEE
ejb3	WFLYEJB
embedded	WFLYEMB
host-controller	WFLYDC
host-controller	WFLYHC
iiop-openjdk	WFLYIIOP
io/subsystem	WFLYIO
jaxrs	WFLYRS
jdr	WFLYJDR
jmx	WFLYJMX
jpa/hibernate5	JIPi
jpa/spi/src/main/java/org/jipijapa/JipiLogger.java	JIPi
jpa/subsystem	WFLYJPA
jsf/subsystem	WFLYJSF
jsr77	WFLYEEMGMT
launcher	WFLYLNCHR
legacy/jacorb	WFLYORB
legacy/messaging	WFLYMSG

Maven モジュール	プロジェクトコード
legacy/web	WFLYWEB
logging	WFLYLOG
mail	WFLYMAIL
management-client-content	WFLYCNT
messaging-activemq	WFLYMSGAMQ
mod_cluster/extension	WFLYMODCLS
naming	WFLYNAM
network	WFLYNET
patching	WFLYPAT
picketlink	WFLYPL
platform-mbean	WFLYPMB
pojo	WFLYPOJO
process-controller	WFLYPC
protocol	WFLYPRT
remoting	WFLYRMT
request-controller	WFLYREQCON
rts	WFLYRTS
sar	WFLYSAR
security-manager	WFLYSM
security	WFLYSEC
server	WFLYSRV
system-jmx	WFLYSYSJMX
threads	WFLYTHR

Maven モジュール	プロジェクトコード
transactions	WFLYTX
undertow	WFLYUT
webservices/server-integration	WFLYWS
weld	WFLYWELD
xts	WFLYXTS

第5章 リモート JNDI ルックアップ

5.1. JNDI へのオブジェクトの登録

Java Naming and Directory Interface (JNDI) は、Java ソフトウェアクライアントによる名前を使ったオブジェクトの検出およびルックアップを可能にするディレクトリーサービスの Java API です。

JNDI に登録されているオブジェクトがリモート JNDI クライアント (別の JVM で実行されるクライアントなど) によってルックアップされる必要がある場合、オブジェクトを **java:jboss/exported** コンテキスト下に登録する必要があります。

たとえば、**messaging-activemq** サブシステムの JMS キューをリモート JNDI クライアントに公開する必要がある場合は、**java:jboss/exported/jms/queue/myTestQueue** を使用して JMS キューを JNDI に登録する必要があります。リモート JNDI クライアントは、名前 **jms/queue/myTestQueue** で JMS キューをルックアップできます。

例: **standalone-full(-ha).xml** のキューの設定

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <jms-queue name="myTestQueue" entries="java:jboss/exported/jms/queue/myTestQueue"/>
    ...
  </server>
</subsystem>
```

5.2. リモート JNDI の設定

リモート JNDI クライアントは接続し、JNDI から名前でオブジェクトを検索できます。リモート JNDI クライアントを使用してオブジェクトを検索するには、**jboss-client.jar** がクラスパスに指定されている必要があります。**jboss-client.jar** は **EAP_HOME/bin/client/jboss-client.jar** で利用できます。

以下の例は、リモート JNDI クライアントの JNDI から **myTestQueue** キューをルックアップする方法を示しています。

例: **MDB リソースアダプターの設定**

```
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
properties.put(Context.PROVIDER_URL, "remote+http://HOST_NAME:8080");
context = new InitialContext(properties);
Queue myTestQueue = (Queue) context.lookup("jms/queue/myTestQueue");
```

5.3. HTTP 上の JNDI 呼び出し

HTTP 上の JNDI 呼び出しには、クライアント側実装とサーバー側実装の 2 つの異なる部分が含まれます。

5.3.1. クライアント側実装

クライアント側実装はリモートネーミング実装と似ていますが、Undertow HTTP クライアントを使用して HTTP を基にしています。

接続管理は直接的ではなく暗黙的で、既存のリモートネーミング実装で使用される方法と似ているキャッシングを使用します。接続プールは、接続パラメーターを基にキャッシュされます。接続プールが指定された期間使用されないと、破棄されます。

HTTP トランスポートを使用するようにリモート JNDI クライアントアプリケーションを設定するには、以下の依存関係を HTTP トランスポート実装に追加する必要があります。

```
<dependency>
  <groupId>org.wildfly.wildfly-http-client</groupId>
  <artifactId>wildfly-http-naming-client</artifactId>
</dependency>
```

HTTP 呼び出しを実行するには、**http** URL スキームを使用し、HTTP インボーカーのコンテキスト名である **wildfly-services** を含める必要があります。たとえば、**remote+http://localhost:8080** をターゲット URL として使用している場合、HTTP トランスポートを使用するにはこれを **http://localhost:8080/wildfly-services** に更新する必要があります。

5.3.2. サーバー側実装

サーバー側実装は既存のリモートネーミング実装と似ていますが、HTTP トランスポートを使用しません。

サーバーを設定するには、**undertow** サブシステムで使用したい各仮想ホストの **http-invoker** を有効にする必要があります。これは、標準の設定ではデフォルトで有効になっています。無効になっている場合は、以下の管理 CLI コマンドを使用して再度有効にします。

```
/subsystem=undertow/server=default-server/host=default-host/setting=http-invoker:add(http-authentication-factory=myfactory, path="/wildfly-services")
```

http-invoker 属性は2つのパラメーターを取ります。その1つは **path** で、デフォルトは **/wildfly-services** になります。もう1つのパラメーターは **http-authentication-factory** で、Elytron の **http-authentication-factory** への参照である必要があります。



注記

http-authentication-factory の使用を希望するすべてのデプロイメントは、Elytron セキュリティと、指定の HTTP 認証ファクトリーに対応する同じセキュリティドメインを使用する必要があります。

第6章 WEB アプリケーションのクラスター化

6.1. セッションレプリケーション

6.1.1. HTTP セッションレプリケーション

セッションレプリケーションは、配布可能なアプリケーションのクライアントセッションが、クラスター内のノードのフェイルオーバーによって中断されないようにします。クラスター内の各ノードは実行中のセッションの情報を共有するため、ノードが消滅してもセッションを引き継ぐことができます。

セッションレプリケーションは、mod_cluster、mod_jk、mod_proxy、ISAPI、および NSAPI クラスターにより高可用性を確保する仕組みのことで、

6.1.2. アプリケーションにおけるセッションレプリケーションの有効化

JBoss EAP の高可用性 (HA) 機能を利用し、Web アプリケーションのクラスターリングを有効にするには、アプリケーションが配布可能になるよう設定する必要があります。アプリケーションが配布可能でないと、そのセッションは配布されません。

アプリケーションを配布可能にする

1. アプリケーションの **web.xml** 記述子ファイルの **<web-app>** タグ内に **<distributable/>** 要素を追加します。

例: 配布可能なアプリケーションの最低限の設定

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_3_0.xsd"
  version="3.0">

  <distributable/>

</web-app>
```

2. 次に、必要な場合はデフォルトのレプリケーションの動作を変更します。セッションレプリケーションに影響する値を変更する場合は、アプリケーションの **WEB-INF/jboss-web.xml** ファイルにある **<jboss-web>** 内の **<replication-config>** 要素内でこれらの値をオーバーライドできます。該当する要素で、デフォルト値をオーバーライドする場合のみ値を含めます。

例: <replication-config> の値

```
<jboss-web xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
    http://www.jboss.org/j2ee/schema/jboss-web_10_0.xsd">
  <replication-config>
    <replication-granularity>SESSION</replication-granularity>
  </replication-config>
</jboss-web>
```

<replication-granularity> パラメーターは、レプリケートされるデータの粒度を決定します。デフォルト値は **SESSION** ですが、**ATTRIBUTE** を設定すると、ほとんどの属性は変更されずにセッションのパフォーマンスを向上させることができます。

<replication-granularity> の有効値は以下のとおりです。

- **SESSION**: デフォルト値です。属性がダーティーである場合に、セッションオブジェクト全体がレプリケートされます。このポリシーは、オブジェクト参照が複数のセッション属性で共有される場合に必要です。共有されるオブジェクト参照は、1つのユニットでセッション全体がシリアライズされるため、リモートノードで維持されます。
- **ATTRIBUTE**: これは、セッションのダーティーな属性と一部のセッションデータ (最後にアクセスされたタイムスタンプなど) にのみに使用できる値です。

変更不能なセッション属性

JBoss EAP 7 では、セッションが変更されたり、セッションの変更可能な属性がアクセスされたときにセッションレプリケーションが発生します。以下のいずれかの条件に該当しない限り、セッション属性は変更可能であると見なされます。

- 値が既知の変更不能な値である:
 - `null`
 - `java.util.Collections.EMPTY_LIST`、`EMPTY_MAP`、`EMPTY_SET`
- 値の型がまたは既知の変更不能な型である、または既知の変更不能な型を実装する:
 - `java.lang.Boolean`、`Character`、`Byte`、`Short`、`Integer`、`Long`、`Float`、`Double`
 - `java.lang.Class`、`Enum`、`StackTraceElement`、`String`
 - `java.io.File`、`java.nio.file.Path`
 - `java.math.BigDecimal`、`BigInteger`、`MathContext`
 - `java.net.Inet4Address`、`Inet6Address`、`InetSocketAddress`、`URI`、`URL`
 - `java.security.Permission`
 - `java.util.Currency`、`Locale`、`TimeZone`、`UUID`
 - `java.time.Clock`、`Duration`、`Instant`、`LocalDate`、`LocalDateTime`、`LocalTime`、`MonthDay`、`Period`、`Year`、`YearMonth`、`ZonedDateTime`、`ZoneOffset`、`ZonedDateTime`
 - `java.time.chrono.ChronoLocalDate`、`Chronology`、`Era`
 - `java.time.format.DateTimeFormatter`、`DecimalStyle`
 - `java.time.temporal.TemporalField`、`TemporalUnit`、`ValueRange`、`WeekFields`
 - `java.time.zone.ZoneOffsetTransition`、`ZoneOffsetTransitionRule`、`ZoneRules`
- 値の型に以下のアノテーションが付けられる:
 - `@org.wildfly.clustering.web.annotation.Immutable`
 - `@net.jcip.annotations.Immutable`

6.2. HTTP セッションパッシベーションおよびアクティベーション

6.2.1. HTTP セッションパッシベーションおよびアクティベーション

パッシベーションとは、比較的利用されていないセッションをメモリーから削除し、永続ストレージへ保存することでメモリーの使用量を制御するプロセスのことです。

アクティベーションとは、パッシベートされたデータを永続ストレージから取得し、メモリーに戻すことです。

パッシベーションは HTTP セッションのライフタイムの異なるタイミングで実行されます。

- コンテナが新規セッションの作成を要求するときに現在アクティブなセッションの数が設定上限を超えている場合、サーバーはセッションの一部をパッシベートして新規セッションのスペースを作成しようとします。
- Web アプリケーションがデプロイされ、他のサーバーでアクティブなセッションのバックアップコピーが、新たにデプロイされる Web アプリケーションのセッションマネージャーによって取得された場合、セッションはパッシベートされることがあります。

アクティブなセッションが設定可能な最大数を超えると、セッションはパッシベートされます。

セッションは常に LRU (Least Recently Used) アルゴリズムを使ってパッシベートされます。

6.2.2. アプリケーションでの HTTP セッションパッシベーションの設定

HTTP セッションパッシベーションは、アプリケーションの **WEB-INF/jboss-web.xml** および **META-INF/jboss-web.xml** ファイルで設定されます。

例: **jboss-web.xml** ファイル

```
<jboss-web xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
  http://www.jboss.org/j2ee/schema/jboss-web_10_0.xsd">

  <max-active-sessions>20</max-active-sessions>
</jboss-web>
```

<max-active-sessions> 要素により、許可されるアクティブなセッションの最大数が設定され、セッションパッシベーションが有効になります。セッションの作成によって、アクティブなセッションの数が **<max-active-sessions>** を超える場合は、新しいセッションのスペースを確保するために、セッションマネージャーが認識する最も古いセッションがパッシベートされます。



注記

メモリーのセッションの合計数には、このノードでアクセスされていない他のクラスターノードからレプリケートされたセッションが含まれます。これを考慮して **<max-active-sessions>** を設定してください。また、他のノードからレプリケートされるセッションの数は、**REPL** または **DIST** キャッシュモードが有効であるかどうかによっても異なります。**REPL** キャッシュモードでは、各セッションは各ノードにレプリケートされます。**DIST** キャッシュモードでは、各セッションは、**owners** パラメーターによって指定された数のノードにのみレプリケートされます。セッションキャッシュモードの設定は、JBoss EAP 『[設定ガイド](#)』の「[キャッシュモードの設定](#)」を参照してください。たとえば、各ノードが100ユーザーからの要求を処理する8つのノードクラスターがあるとします。この場合、**REPL** キャッシュモードでは、各ノードのメモリーに800のセッションが格納されます。**DIST** キャッシュモードが有効であり、デフォルトの **owners** 設定が2であるときは、各ノードのメモリーには200のセッションが格納されます。

6.3. クラスタリングサービスのパブリック API

JBoss EAP 7 には、アプリケーションが使用する改善されたパブリッククラスタリング API が導入されました。新しいサービスは、ライトウェイトで簡単に挿入できるよう設計されています。外部の依存関係は必要ありません。

org.wildfly.clustering.group.Group

グループサービスは、JGroups チャンネルのクラスタートポロジを参照し、トポロジの変更時に通知するメカニズムを提供します。

```
@Resource(lookup = "java:jboss/clustering/group/channel-name")
private Group channelGroup;
```

org.wildfly.clustering.dispatcher.CommandDispatcher

CommandDispatcherFactory サービスは、クラスター内のノードでコマンドを実行するためにディスパッチャーを作成するメカニズムを提供します。結果として得られる **CommandDispatcher** は、以前の JBoss EAP リリースのリフレクションベースの **GroupRpcDispatcher** に類似したコマンドパターンです。

```
@Resource(lookup = "java:jboss/clustering/dispatcher/channel-name")
private CommandDispatcherFactory factory;

public void foo() {
    String context = "Hello world!";
    // Exclude node1 and node3 from the executeOnCluster
    try (CommandDispatcher<String> dispatcher = this.factory.createCommandDispatcher(context))
    {
        dispatcher.executeOnGroup(new StdOutCommand(), node1, node3);
    }
}

public static class StdOutCommand implements Command<Void, String> {
    @Override
    public Void execute(String context) {
        System.out.println(context);
        return null;
    }
}
```

6.4. HA シングルトンサービス

クラスター化されたシングルトンサービス (高可用性 (HA) シングルトンとも呼ばれます) は、クラスターの複数のノードにデプロイされるサービスです。このサービスはいずれかのノードでのみ提供されます。シングルトンサービスが実行されているノードは、通常**マスターノード**と呼ばれます。

マスターノードが失敗またはシャットダウンした場合に、残りのノードから別のマスターが選択され、新しいマスターでサービスが再開されます。マスターが停止し、別のマスターが引き継ぐまでの短い間を除き、サービスは1つのノードのみによって提供されます。

HA シングルトン ServiceBuilder API

JBoss EAP 7 では、プロセスを大幅に簡略化するシングルトンサービスを構築するための新しいパブリック API が導入されました。

SingletonServiceBuilder 実装により、サービスは非同期的に開始されるようインストールされ、Modular Service Container (MSC) のデッドロックが回避されます。

HA シングルトンサービス選択ポリシー

HA シングルトンを起動するノードの優先順位がある場合は、**ServiceActivator** クラスで選択ポリシーを設定できます。

JBoss EAP は、以下の2つの選択ポリシーを提供します。

- 単純な選択ポリシー

単純な選択ポリシーでは、相対的な経過時間に基づいてマスターノードが選択されます。必要な経過時間は、利用可能なノードのリストのインデックスである `position` プロパティで、以下のように設定されます。

 - `position = 0` 最も古いノードを参照します。これはデフォルトです。
 - `position = 12` 番目に古いノードを参照します。

`position` を負の値にして最も新しいノードを示すこともできます。

 - `position = -1` 最も新しいノードを参照します。
 - `position = -22` 番目に新しいノードを参照します。
- ランダムな選択ポリシー

ランダムな選択ポリシーでは、シングルトンサービスのプロバイダーとなるランダムなメンバーが選択されます。

HA シングルトンサービスの優先度

HA シングルトンサービスの選択ポリシーは、任意で1つ以上の優先されるサーバーを指定することができます。優先されるサーバーが利用できる場合は、そのポリシーにおけるすべてのシングルトンアプリケーションでそのサーバーがマスターになります。

優先度は、ノード名またはアウトバウンドソケットバインディング名を介して定義することができます。



注記

ノードの優先度は常に選択ポリシーの結果よりも優先されます。

デフォルトでは、JBoss EAP の高可用性設定によって、優先されるサーバーがない **default** という名前の簡単な選択ポリシーが提供されます。優先度を設定するには、カスタムポリシーを作成し、優先されるサーバーを定義します。

クォーラム

ネットワークパーティションがある場合、シングルトンサービスに潜在的な問題が存在します。この問題はスプリットブレインと呼ばれ、ノードの各サブセットは他のサブセットと通信できません。サーバーの各セットは他のセットのすべてのサーバーに障害が発生したと見なし、唯一障害が発生していないクラスターとして動作します。これによってデータの整合性に問題が発生する可能性があります。

JBoss EAP では、選択ポリシーでクォーラムを指定してスプリットブレインを防ぐことができます。クォーラムは、シングルトンプロバイダーの選択が行われる前に存在するノードの最小数を指定します。

典型的なデプロイメントでは、 $N/2 + 1$ をクォーラムとして使用します。N は予想されるクラスターサイズになります。この値は実行時に更新でき、アクティブなすべてのシングルトンサービスに即時反映されます。

HA シングルトンサービスアプリケーションの作成

以下に、アプリケーションを作成し、クラスター全体のシングルトンサービスとしてデプロイするのに必要な手順の簡単な例を示します。この例では、頻繁にシングルトンサービスをクエリーし、そのシングルトンサービスが実行されているノードの名前を取得します。

シングルトンの挙動を確認するには、最低でも 2 つのサーバーにアプリケーションをデプロイする必要があります。シングルトンサービスが同じノードで実行されているかまたは値がリモートで取得されているかは透過的です。

1. **SingletonService** クラスを作成します。クエリーサービスによって呼び出される **getValue()** メソッドは、実行されているノードに関する情報を提供します。

```
class SingletonService implements Service<Node> {
    private Logger LOG = Logger.getLogger(this.getClass());
    private InjectedValue<Group> group;

    SingletonService(InjectedValue<Group> group) {
        this.group = group;
    }

    @Override
    public void start(StartContext context) throws StartException {
        LOG.infof("Singleton service is starting on node '%s'",
this.group.getValue().getLocalNode());
    }

    @Override
    public void stop(StopContext context) {
        LOG.infof("Singleton service is stopping on node '%s'",
this.group.getValue().getLocalNode());
    }

    @Override
    public Node getValue() throws IllegalStateException, IllegalArgumentException {
        return this.group.getValue().getLocalNode();
    }
}
```

- クエリーサービスを作成します。シングルトンサービスの `getValue()` メソッドを呼び出し、それが稼働しているノードの名前を取得して、サーバーログに結果を書き込みます。

```
class QueryingService implements Service<Void> {

    private Logger LOG = Logger.getLogger(this.getClass());
    private ScheduledExecutorService executor;

    @Override
    public void start(StartContext context) throws StartException {
        LOG.info("Querying service is starting.");

        executor = Executors.newSingleThreadScheduledExecutor();
        executor.scheduleAtFixedRate(() -> {

            @SuppressWarnings("unchecked")
            ServiceController<Node> service = (ServiceController<Node>)
context.getController().getServiceContainer()
                .getService(ServiceActivator.SINGLETON_SERVICE_NAME);
            try {
                Node node = service.awaitValue(5, TimeUnit.SECONDS);
                LOG.infof("Singleton service is running on node '%s'", node);
            } catch (InterruptedException | TimeoutException | IllegalStateException e) {
                LOG.warn("Failed to query singleton service.");
            }

        }, 5, 5, TimeUnit.SECONDS);
    }

    @Override
    public void stop(StopContext context) {
        LOG.info("Querying service is stopping.");

        executor.shutdown();
    }

    @Override
    public Void getValue() throws IllegalStateException, IllegalArgumentException {
        return null;
    }
}
```

- ServiceActivator** クラスを実装し、シングルトンサービスとクエリーサービスの両方を構築およびインストールします。

```
public class ServiceActivator implements org.jboss.msc.service.ServiceActivator {

    private final Logger LOG = Logger.getLogger(ServiceActivator.class);
    static final ServiceName SINGLETON_SERVICE_NAME =
        ServiceName.parse("org.jboss.as.quickstarts.ha.singleton.service.primary-only");
    private static final ServiceName QUERYING_SERVICE_NAME =
        ServiceName.parse("org.jboss.as.quickstarts.ha.singleton.service.primary-
only.querying");

    @Override
```

```

public void activate(ServiceActivatorContext serviceActivatorContext) {
    try {
        SingletonPolicy policy = (SingletonPolicy) serviceActivatorContext
            .getServiceRegistry()

        .getRequiredService(ServiceName.parse(SingletonDefaultRequirement.SINGLETON_POLICY
        .getName()))
            .awaitValue();

        InjectedValue<Group> group = new InjectedValue<>();

        Service<Node> service = new SingletonService(group);

        policy.createSingletonServiceBuilder(SINGLETON_SERVICE_NAME, service)
            .build(serviceActivatorContext.getServiceTarget())
            .addDependency(ServiceName.parse("org.wildfly.clustering.default-group"),
            Group.class, group)
            .install();

        serviceActivatorContext.getServiceTarget()
            .addService(QUERYING_SERVICE_NAME, new QueryingService())
            .setInitialMode(ServiceController.Mode.ACTIVE)
            .install();

        LOG.info("Singleton and querying services activated.");
    } catch (InterruptedException e) {
        throw new ServiceRegistryException(e);
    }
}
}
}

```

4. **ServiceActivator** クラスの名前 (例: **org.jboss.as.quickstarts.ha.singleton.service.primary.ServiceActivator**) が含まれる、**org.jboss.msc.service.ServiceActivator** という名前のファイルを **META-INF/services/** ディレクトリーに作成します。

完全な作業例は、JBoss EAP に同梱される **ha-singleton-service** クイックスタートを参照してください。このクイックスタートには、バックアップサービスでインストールされるシングルトンサービスを実証する2つ目の例も含まれています。バックアップサービスは、シングルトンサービスの実行用に選択されていないすべてのノード上で実行されます。また、このクイックスタートは異なる選択ポリシーの設定方法についても実証します。

6.5. HA シングルトンデプロイメント

JBoss EAP 7 には、該当するアプリケーションをシングルトンデプロイメントとしてデプロイする機能が追加されます。

クラスタ化されたサーバーのグループにデプロイされる場合、シングルトンデプロイメントでは該当するタイミングで単一のノードにのみデプロイされます。デプロイメントがアクティブなノードが停止または失敗すると、デプロイメントは別のノードで自動的に開始されます。

HA シングルトンの動作を制御するポリシーは、新しい **singleton** サブシステムによって管理されます。デプロイメントは特定のシングルトンポリシーを指定するか、デフォルトのサブシステムポリシーを使用します。デプロイメントは、デプロイメントオーバーレイとして既存のデプロイメントに最も簡

単に適用される **META-INF/singleton-deployment.xml** デプロイメント記述子を使用して、シングルTONデプロイメントとして識別されます。また、必要なシングルTON設定を既存の **jboss-all.xml** ファイル内に組み込むこともできます。

シングルTONデプロイメントの定義または選択

- デプロイメントをシングルTONデプロイメントとして定義するには、アプリケーションアーカイブに **META-INF/singleton-deployment.xml** 記述子を含めます。

例: シングルTONデプロイメント記述子

```
<?xml version="1.0" encoding="UTF-8"?>
<singleton-deployment xmlns="urn:jboss:singleton-deployment:1.0"/>
```

例: 特定のシングルTONポリシーを使用したシングルTONデプロイメント記述子

```
<?xml version="1.0" encoding="UTF-8"?>
<singleton-deployment policy="my-new-policy" xmlns="urn:jboss:singleton-deployment:1.0"/>
```

- または、**singleton-deployment** 要素を **jboss-all.xml** 記述子ファイルに追加することもできます。

例: jboss-all.xml での singleton-deployment の定義

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss xmlns="urn:jboss:1.0">
  <singleton-deployment xmlns="urn:jboss:singleton-deployment:1.0"/>
</jboss>
```

例: 特定のシングルTONポリシーを使用した jboss-all.xml での singleton-deployment の定義

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss xmlns="urn:jboss:1.0">
  <singleton-deployment policy="my-new-policy" xmlns="urn:jboss:singleton-deployment:1.0"/>
</jboss>
```

シングルTONデプロイメントの作成

JBoss EAP は、以下の 2 つの選択ポリシーを提供します。

- 単純な選択ポリシー
simple-election-policy は **position** 属性で示された特定のメンバーを選択します (該当するアプリケーションがデプロイされます)。**position** 属性は、降順の経過時間でソートされた候補のリストから選択するノードのインデックスを決定します (**0** は最も古いノード、**1** は 2 番目に古いノード、**-1** は最も新しいノード、**-2** は 2 番目に新しいノードを示します)。指定された位置が候補の数を超えると、モジュロ演算が適用されます。

例: 管理 CLI を使用して simple-election-policy で新しいシングルTONポリシーを作成し、位置を -1 に設定

```
batch
/subsystem=singleton/singleton-policy=my-new-policy:add(cache-container=server)
```

```
/subsystem=singleton/singleton-policy=my-new-policy/election-
policy=simple:add(position=-1)
run-batch
```



注記

新しく作成されたポリシー **my-new-policy** をデフォルトとして設定するには、以下のコマンドを実行します。

```
/subsystem=singleton:write-attribute(name=default, value=my-new-policy)
```

例: standalone-ha.xml で位置を -1 として simple-election-policy を設定

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
  <singleton-policies default="my-new-policy">
    <singleton-policy name="my-new-policy" cache-container="server">
      <simple-election-policy position="-1"/>
    </singleton-policy>
  </singleton-policies>
</subsystem>
```

- ランダムな選択ポリシー

random-election-policy は、該当するアプリケーションがデプロイされるランダムなメンバーを選択します。

例: 管理 CLI を使用して random-election-policy で新しいシングルトンポリシーを作成

```
batch
/subsystem=singleton/singleton-policy=my-other-new-policy:add(cache-container=server)
/subsystem=singleton/singleton-policy=my-other-new-policy/election-policy=random:add()
run-batch
```

例: standalone-ha.xml を使用した random-election-policy の設定

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
  <singleton-policies default="my-other-new-policy">
    <singleton-policy name="my-other-new-policy" cache-container="server">
      <random-election-policy/>
    </singleton-policy>
  </singleton-policies>
</subsystem>
```



注記

ポリシーを追加する前に、**cache-container** の **default-cache** 属性を定義する必要があります。定義しないと、カスタムキャッシュコンテナを使用する場合に、エラーメッセージが表示されることがあります。

また、シングルトン選択ポリシーを使用して、クラスターの1つ以上のメンバーの優先順位を指定することもできます。優先順位は、ノード名またはアウトバウンドソケットバインド名を使用して定義できます。ノードの優先順位は、常に選択ポリシーの結果よりも優先されます。

例: 管理 CLI を使用した既存のシングルトンポリシーの優先順位の指定

```
/subsystem=singleton/singleton-policy=foo/election-policy=simple:list-add(name=name-preferences,
value=nodeA)

/subsystem=singleton/singleton-policy=bar/election-policy=random:list-add(name=socket-binding-
preferences, value=binding1)
```

例: 管理 CLI を使用して simple-election-policy および name-preferences で新しいシングルトンポリシーを設定

```
batch
/subsystem=singleton/singleton-policy=my-new-policy:add(cache-container=server)
/subsystem=singleton/singleton-policy=my-new-policy/election-policy=simple:add(name-preferences=
[node1, node2, node3, node4])
run-batch
```



注記

新しく作成されたポリシー **my-new-policy** をデフォルトとして設定するには、以下のコマンドを実行します。

```
/subsystem=singleton:write-attribute(name=default, value=my-new-policy)
```

例: standalone-ha.xml を使用した socket-binding-preferences での random-election-policy の設定

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
  <singleton-policies default="my-other-new-policy">
    <singleton-policy name="my-other-new-policy" cache-container="server">
      <random-election-policy>
        <socket-binding-preferences>binding1 binding2 binding3 binding4</socket-binding-
preferences>
      </random-election-policy>
    </singleton-policy>
  </singleton-policies>
</subsystem>
```

クォーラムの定義

ネットワークパーティションは、シングルトンデプロイメントに対して特に問題となります。これは、同時に実行する同じデプロイメントに対して複数のシングルトンプロバイダーをトリガーできるためです。このような状況を回避するために、シングルトンポリシーはシングルトンプロバイダーの選択が行われる前に、最小数のノードが存在することを必要とするクォーラムを定義できます。典型的なデプロイメントでは、 $N/2 + 1$ をクォーラムとして使用します (ここで、 N は予想されるクラスターサイズです)。この値は実行時に更新でき、対応するシングルトンポリシーを使用するすべてのシングルトンサービスに即時反映されます。

例: standalone-ha.xml ファイルでのクォーラム宣言

-

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
  <singleton-policies default="default">
    <singleton-policy name="default" cache-container="server" quorum="4">
      <simple-election-policy/>
    </singleton-policy>
  </singleton-policies>
</subsystem>
```

例: 管理 CLI を使用したクォーラム宣言

```
/subsystem=singleton/singleton-policy=foo:write-attribute(name=quorum, value=3)
```

シングルトンデプロイメントを使用したクラスター全体のシングルトンとしてアプリケーションにパッケージ化されたサービスの完全な作業例は、JBoss EAP に同梱される **ha-singleton-deployment** クイックスタートを参照してください。

6.6. APACHE MOD_CLUSTER-MANAGER アプリケーション

6.6.1. mod_cluster-manager アプリケーション

mod_cluster-manager アプリケーションは、Apache HTTP サーバーで利用可能な管理 Web ページです。接続されたワーカーノードを監視し、コンテキストの有効化または無効化やクラスター内のワーカーノードの負荷分散プロパティの設定などのさまざまな管理タスクを実行するために使用されます。

mod_cluster-manager アプリケーションの使用

mod_cluster-manager アプリケーションは、ワーカーノードでさまざまな管理タスクを実行するために使用されます。

mod_cluster/1.3.1.Final 1

[Auto Refresh](#) [show DUMP output](#) [show INFO output](#)

LBGroup Group-EU-North: [Enable Nodes](#) [Disable Nodes](#) [Stop Nodes](#)

Node jboss-eap-7.0-3 (ajp://192.168.122.172:8211): 2

[Enable Contexts](#) [Disable Contexts](#) [Stop Contexts](#) 7

Balancer: qacluster:LBGroup: Group-EU-North.Flushpackets: Off.Flushwait: 10000.Ping: 10000000.Smax: 2.Ttl: 60000000.Status: OK.Elected: 10.Read: 5960.Transferred: 0.Connected: 0.Load: 73

Virtual Host 1: 4

Contexts:

/clusterbench, Status: ENABLED Request: 0 **5 6** [Disable](#) [Stop](#)

Aliases:

default-host
localhost

LBGroup Group-EU-West: [Enable Nodes](#) [Disable Nodes](#) [Stop Nodes](#)

Node jboss-eap-7.0-2 (ajp://192.168.122.172:8110): 3

[Enable Contexts](#) [Disable Contexts](#) [Stop Contexts](#)

Balancer: qacluster:LBGroup: Group-EU-West.Flushpackets: Off.Flushwait: 10000.Ping: 10000000.Smax: 2.Ttl: 60000000.Status: OK.Elected: 1.Read: 593.Transferred: 0.Connected: 0.**Load: 73**

Virtual Host 1: 8 9

Contexts:

/clusterbench, Status: ENABLED Request: 0 [Disable](#) [Stop](#)

Aliases:

localhost
default-host

図 - mod_cluster 管理 Web ページ

- [1] mod_cluster/1.3.1.Final: mod_cluster ネイティブライブラリーのバージョン。

- [2] `ajp://192.168.122.204:8099`: 使用されるプロトコル (AJP、HTTP、または HTTPS)、ワーカーノードのホスト名または IP アドレス、およびポート。
- [3] `jboss-eap-7.0-2`: ワーカーノードの `JVMRoute`。
- [4] **Virtual Host 1**: ワーカーノードで設定された仮想ホスト。
- [5] **Disable**: 特定のコンテキストで新しいセッションの作成を無効にするために使用できる管理オプション。ただし、現在のセッションは無効にされず、そのまま処理されます。
- [6] **Stop**: コンテキストへのセッション要求のルーティングを停止するために使用できる管理オプション。**sticky-session-force** プロパティが **true** に設定されない限り、残りのセッションは別のノードにフェイルオーバーされます。
- [7] **Enable Contexts Disable Contexts Stop Contexts** ノード全体で実行できる操作。これらのいずれかのオプションを選択すると、すべての仮想ホストのノードのコンテキストすべてが影響を受けます。
- [8] **Load balancing group (LBGroup)**: すべてのワーカーノードをカスタム負荷分散グループにグループ化するために、**load-balancing-group** プロパティは JBoss EAP 設定の **modcluster** サブシステムで設定されます。負荷分散グループ (LBGroup) は、設定されたすべての負荷分散グループに関する情報を提供する情報フィールドです。このフィールドが設定されていないと、すべてのワーカーノードは単一のデフォルト負荷分散グループにグループ化されます。



注記

これは唯一の情報フィールドであるため、**load-balancing-group** プロパティの設定に使用できません。このプロパティは JBoss EAP 設定の **modcluster** サブシステムで設定する必要があります。

- [9] **Load (value)**: ワーカーノードの負荷係数。負荷係数は以下のように評価されます。

-load > 0 : A load factor with value 1 indicates that the worker node is overloaded. A load factor of 100 denotes a free and not-loaded node.
 -load = 0 : A load factor of value 0 indicates that the worker node is in standby mode. This means that no session requests will be routed to this node until and unless the other worker nodes are unavailable.
 -load = -1 : A load factor of value -1 indicates that the worker node is in an error state.
 -load = -2 : A load factor of value -2 indicates that the worker node is undergoing CPing/CPong and is in a transition state.



注記

JBoss EAP 7.2 の場合は、ロードバランサーとして Undertow を使用することもできます。

第7章 コンテキストおよび依存関係の挿入 (CDI)

7.1. CDI の概要

7.1.1. コンテキストと依存関係の注入 (CDI)

Contexts and Dependency Injection (CDI) 2.0 は、Enterprise Java Beans (EJB) 3 コンポーネントを Java Server Faces (JSF) 管理対象 Bean として使用できるよう設計された仕様です。2つのコンポーネントモデルを統合し、Java を使用した Web ベースのアプリケーション向けプログラミングモデルを大幅に簡略化します。CDI 2.0 の詳細は、『[JSR 365: Contexts and Dependency Injection for Java 2.0](#)』を参照してください。

JBoss EAP には、JSR-365 のリファレンス実装である Weld が含まれています。

CDI の利点

CDI には以下のような利点があります。

- 多くのコードをアノテーションに置き換えることにより、コードベースが単純化および削減されます。
- 柔軟であり、インジェクションおよびイベントを無効または有効にしたり、代替の Bean を使用したり、非 CDI オブジェクトを簡単にインジェクトしたりできます。
- デフォルトとは異なる設定をカスタマイズする必要がある場合、任意で **beans.xml** ファイルを **META-INF/** または **WEB-INF/** ディレクトリーに含めることができます。ファイルは空にすることができます。
- パッケージ化とデプロイメントが簡略化され、デプロイメントに追加する必要がある XML の量が減少します。
- コンテキストを使用したライフサイクル管理が提供されます。インジェクションを要求、セッション、会話、またはカスタムコンテキストに割り当てることができます。
- 文字列ベースのインジェクションよりも安全かつ簡単にデバッグを行える、タイプセーフな依存関係の注入が提供されます。
- インターセプターと Bean が切り離されます。
- 複雑なイベント通知が提供されます。

7.1.2. Weld、Seam 2、および JavaServer Faces 間の関係

Weld は [JSR 346: Contexts and Dependency Injection for Java™ EE 1.1](#) で定義されている CDI の参照実装です。Weld は、Seam 2 と他の依存関係注入フレームワークの影響を受けており、JBoss EAP 6 に含まれています。

Seam 2 の目的は、Enterprise Java Bean と JavaServer Faces 管理対象 Bean を統合することでした。

JavaServer Faces 2.2 では、[JSR-344: JavaServer™ Faces 2.2](#) が実装されます。これは、サーバーサイドユーザーインターフェースをビルドするための API です。

7.2. CDI を使用したアプリケーションの開発

コンテキストと依存関係の注入 (CDI: Contexts and Dependency Injection) を使用すると、アプリケー

ションの開発、コードの再利用、デプロイメント時または実行時のコードの調整、およびユニットテストを非常に柔軟に行うことができます。JBoss EAP には、CDI の参照実装である Weld が含まれます。これらのタスクは、エンタープライズアプリケーションで CDI を使用方法を示しています。

Weld にはアプリケーション開発の特別なモードが含まれています。開発モードを有効にすると、CDI アプリケーションの開発を容易にする一部のビルドインツールが利用できます。



注記

アプリケーションのパフォーマンスに悪影響を与えるため、開発モードは本番環境では使用しないでください。必ずデプロイメントモードを無効にしてから本番環境にデプロイしてください。

Web アプリケーションに対して開発モードを有効にするには、以下を行います。

Web アプリケーションの場合、サーブレット初期化パラメーター `org.jboss.weld.development` を `true` に設定します。

```
<web-app>
  <context-param>
    <param-name>org.jboss.weld.development</param-name>
    <param-value>true</param-value>
  </context-param>
</web-app>
```

管理 CLI を使用して JBoss EAP に対して開発モードを有効にするには、以下を行います。

`development-mode` 属性を `true` に設定すると、デプロイされたアプリケーションすべてに対して Weld 開発モードをグローバルに有効にすることが可能です。

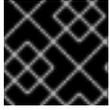
```
/subsystem=weld:write-attribute(name=development-mode,value=true)
```

7.2.1. デフォルトの Bean 検出モード

bean アーカイブのデフォルトの bean 検出モードは **annotated** です。このような bean アーカイブは **implicit bean archive** と呼ばれます。

bean 検出モードが **annotated** の場合:

- **bean defining annotation** がなく、セッション bean の bean クラスでない bean クラスが検出されません。
- セッション bean 上になく、bean クラスが bean 定義アノテーションを持たないプロデューサーメソッドが検出されません。
- セッション bean 上になく、bean クラスが bean 定義アノテーションを持たないプロデューサーフィールドが検出されません。
- セッション bean 上になく、bean クラスが bean 定義アノテーションを持たないディスパーザーメソッドが検出されません。
- セッション bean 上になく、bean クラスが bean 定義アノテーションを持たないオブザーバーメソッドが検出されません。



重要

CDI セクションのすべての例は、検出モードが **all** に設定された場合にのみ有効です。

bean 定義アノテーション

bean クラスは **bean defining annotation** を持つことがあります、bean アーカイブで定義されたようにアプリケーションのどこにでも配置することができます。bean 定義アノテーションを持つ bean クラスは暗黙的な bean と呼ばれます。

bean 定義アノテーションのセットには以下のものが含まれます。

- **@ApplicationScoped**、**@SessionScoped**、**@ConversationScoped** および **@RequestScoped** アノテーション
- その他すべての通常スコープタイプ。
- **@Interceptor** および **@Decorator** アノテーション。
- **@Stereotype** 付けられたアノテーションなど、stereotype アノテーションすべて。
- **@Dependent** スコープアノテーション。

これらのアノテーションのいずれかが bean クラスで宣言された場合、その bean クラスは bean 定義アノテーションを持っていることになります。

例: bean 定義アノテーション

```

@Dependent
public class BookShop
    extends Business
    implements Shop<Book> {
    ...
}

```



注記

他の [JSR-330](#) 実装との互換性を確保するために、**@Dependent** を除くすべての pseudo-scope アノテーションは bean 定義アノテーションではありません。ただし、pseudo-scope アノテーションを含む stereotype アノテーションは bean 定義アノテーションです。

7.2.2. スキャンプロセスからの Bean の除外

除外フィルターは、Bean アーカイブの **beans.xml** ファイルの **<exclude>** 要素によって **<scan>** 要素の子として定義されます。デフォルトでは、除外フィルターはアクティブです。定義に以下のものが含まれる場合、除外フィルターは非アクティブになります。

- **name** 属性を含む、**<if-class-available>** という名前の子要素。Bean アーカイブのクラスローダーはこの名前のクラスをロードできません。
- **name** 属性を含む、**<if-class-not-available>** という名前の子要素。Bean アーカイブのクラスローダーはこの名前のクラスをロードできます。
- **name** 属性を含む、**<if-system-property>** という名前の子要素。この名前に対して定義されたシステムプロパティはありません。

- **name** 属性と値属性を含む、**<if-system-property>** という名前の子要素。この名前とこの値に対して定義されたシステムプロパティはありません。

フィルターがアクティブな場合、タイプは検出から除外され、以下のいずれかの状態になります。

- 検出されるタイプの完全修飾名が、除外フィルターの名前属性の値に一致します。
- 検出されるタイプのパッケージ名が、除外フィルターの接尾辞 **"*"** を含む名前属性の値に一致します。
- 検出されるタイプのパッケージ名が、除外フィルターの接尾辞 **"*"** を含む名前属性の値で始まります。

例7.1 例: beans.xml ファイル

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee">

  <scan>
    <exclude name="com.acme.rest.*" /> ①

    <exclude name="com.acme.faces.**"> ②
      <if-class-not-available name="javax.faces.context.FacesContext"/>
    </exclude>

    <exclude name="com.acme.verbose.*"> ③
      <if-system-property name="verbosity" value="low"/>
    </exclude>

    <exclude name="com.acme.ejb.**"> ④
      <if-class-available name="javax.enterprise.inject.Model"/>
      <if-system-property name="exclude-ejbs"/>
    </exclude>
  </scan>
</beans>
```

- ① 最初の除外フィルターにより、**com.acme.rest** パッケージ内のすべてのクラスが除外されます。
- ② 2番目の除外フィルターにより、**com.acme.faces** パッケージとすべてのサブパッケージ内のすべてのクラスが除外されます (JSF が利用可能でない場合のみ)。
- ③ 3番目の除外フィルターにより、システムプロパティ **verbosity** が値 **low** を持つ場合に、**com.acme.verbose** パッケージ内のすべてのクラスが除外されます。
- ④ 4番目の除外フィルターにより、システムプロパティ **exclude-ejbs** が任意の値で設定され、**javax.enterprise.inject.Model** クラスがクラスローダーでも利用可能な場合に、**com.acme.ejb** パッケージとすべてのサブパッケージ内のすべてのクラスが除外されます。



注記

Java EE コンポーネントに **@Vetoed** アノテーションを付けて Java EE コンポーネントが Bean と見なされないようにすることができます。イベントは **@Vetoed** アノテーションが付けられたタイプに対して実行されず、また **@Vetoed** アノテーションが付けられたパッケージでは実行されません。詳細は「**@Vetoed**」を参照してください。

7.2.3. インジェクションを使用した実装の拡張

インジェクションを使用して、既存のコードの機能を追加または変更できます。

この例では、既存のクラスに翻訳機能を追加します。このメソッド **buildPhrase** を持つ **Welcome** クラスがすでにあることを前提とします。**buildPhrase** メソッドは、都市の名前を引数として取得し、「Welcome to Boston!」などのフレーズを出力します。

この例では、想像上の **Translator** オブジェクトが **Welcome** クラスにインジェクトされます。**Translator** オブジェクトは、文のある言語から別の言語に翻訳できる EJB ステートレス Bean または別のタイプの Bean になります。この例では、**Translator** は挨拶全体を翻訳するために使用され、元の **Welcome** クラスは変更されません。**Translator** は、**buildPhrase** メソッドが呼び出される前にインジェクトされます。

例: Translator bean の Welcome クラスへのインジェクト

```
public class TranslatingWelcome extends Welcome {
    @Inject Translator translator;

    public String buildPhrase(String city) {
        return translator.translate("Welcome to " + city + "!");
    }
    ...
}
```

7.3. あいまいな依存関係または満たされていない依存関係

コンテナーが1つの Bean への注入を解決できない場合、依存関係があいまいとなります。

コンテナーがいずれの Bean に対しても注入の解決をできない場合、依存関係が満たされなくなります。

コンテナーは以下の手順を踏み、依存関係の解決をはかります。

1. インジェクションポイントの Bean 型を実装する全 Bean にある修飾子アノテーションを解決します。
2. 無効となっている Bean をフィルタリングします。無効な Bean とは、明示的に有効化されていない **@Alternative** Bean のことです。

依存関係があいまいな場合、あるいは満たされない場合は、コンテナーはデプロイメントを中断して例外を発生させます。

あいまいな依存関係を修正するには、「[修飾子を使用したあいまいなインジェクションの解決](#)」を参照してください。

7.3.1. 修飾子

修飾子は、コンテナが複数の Bean を解決できるときにあいまいな依存関係を回避するために使用されるアノテーションであり、インジェクションポイントに含まれます。インジェクションポイントで宣言された修飾子は、同じ修飾子を宣言する有効な Bean セットを提供します。

修飾子は、以下の例で示されたように Retention と Target を使用して宣言する必要があります。

例: @Synchronous 修飾子と @Asynchronous 修飾子の定義

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Synchronous {}
```

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Asynchronous {}
```

例: @Synchronous 修飾子と @Asynchronous 修飾子の使用

```
@Synchronous
public class SynchronousPaymentProcessor implements PaymentProcessor {
    public void process(Payment payment) { ... }
}
```

```
@Asynchronous
public class AsynchronousPaymentProcessor implements PaymentProcessor {
    public void process(Payment payment) { ... }
}
```

'@Any'

Bean またはインジェクションポイントにより修飾子が明示的に宣言されない場合、コンテナにより修飾子は **@Default** と見なされます。場合によっては、修飾子を指定せずにインジェクションポイントを宣言する必要があります。この場合も修飾子が存在します。すべての Bean には修飾子 **@Any** が含まれます。したがって、インジェクションポイントで **@Any** を明示的に指定することにより、インジェクションが可能な Bean を制限せずにデフォルトの修飾子を抑制できます。

これは、特定の bean タイプを持つすべての Bean に対して繰り返し処理を行う場合に特に役に立ちます。

```
import javax.enterprise.inject.Instance;
...

@Inject

void initServices(@Any Instance<Service> services) {

    for (Service service: services) {

        service.init();
    }
}
```

```
}
```

```
}
```

各 Bean は、**@Any** 修飾子を明示的に宣言しない場合でもこの修飾子を持ちます。

各イベントも、**@Any** 修飾子が明示的に宣言されずに発生した場合でもこの修飾子を持ちます。

```
@Inject @Any Event<User> anyUserEvent;
```

@Any 修飾子により、インジェクションポイントが特定の Bean タイプのすべての Bean またはイベントを参照できます。

```
@Inject @Delegate @Any Logger logger;
```

7.3.2. 修飾子を使用したあいまいなインジェクションの解決

修飾子を使用してあいまいなインジェクションを解決できます。あいまいなインジェクションは「[あいまいな依存関係または満たされていない依存関係](#)」をお読みください。

以下の例はあいまいであり、**Welcome** の2つの実装 (翻訳を行う1つと翻訳を行わないもう1つ) を含みます。翻訳を行う **Welcome** を使用するには、インジェクションを指定する必要があります。

例: あいまいなインジェクション

```
public class Greeter {
    private Welcome welcome;

    @Inject
    void init(Welcome welcome) {
        this.welcome = welcome;
    }
    ...
}
```

修飾子を使用したあいまいなインジェクションの解決

1. あいまいなインジェクションを解決するには、**@Translating** という名前の修飾子アノテーションを作成します。

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE,METHOD,FIELD,PARAMETERS})
public @interface Translating{}
```

2. 翻訳を行う **Welcome** に **@Translating** アノテーションを付けます。

```
@Translating
public class TranslatingWelcome extends Welcome {
    @Inject Translator translator;
    public String buildPhrase(String city) {
        return translator.translate("Welcome to " + city + "!");
    }
}
```

```

    }
    ...
}

```

- インジェクションで翻訳を行う **Welcome** を要求します。ファクトリーメソッドパターンの場合と同様に、修飾された実装を明示的に要求する必要があります。あいまいさはインジェクションポイントで解決されます。

```

public class Greeter {
    private Welcome welcome;
    @Inject
    void init(@Translating Welcome welcome) {
        this.welcome = welcome;
    }
    public void welcomeVisitors() {
        System.out.println(welcome.buildPhrase("San Francisco"));
    }
}

```

7.4. 管理 BEAN

Java EE では管理対象 Bean 仕様の共通定義が確立されています。管理対象 bean は、プログラミングの制限が最小限であるコンテナ管理オブジェクトとして定義され、POJO (Plain Old Java Object) として知られるようになりました。管理対象 bean はリソースのインジェクション、ライフサイクルコールバック、インターセプターなどの基本サービスの小さなセットをサポートします。EJBや CDI などのコンパニオン仕様は、この基本モデルに基づいて構築されます。

ごくわずかな例外を除き、パラメーターのないコンストラクター (または **@Inject** アノテーションが指定されたコンストラクター) を持つ具象 Java クラスは bean になります。これには、すべての Java Bean と EJB セッション bean が含まれます。

7.4.1. Bean であるクラスのタイプ

管理対象 bean は Java クラスです。管理対象 bean の基本的なライフサイクルやセマンティクスは、管理対象 Bean の仕様で定義されています。bean クラス **@ManagedBean** アノテーションを付けることで明示的に管理対象 bean を宣言できますが、CDI ではその必要はありません。この仕様によると、CDI コンテナでは、以下の条件を満たすクラスはすべて管理対象 bean として扱われます。

- 非静的な内部クラスではないこと。
- 具象クラス、あるいは **@Decorator** アノテーションが付与されている。
- EJB コンポーネントを定義するアノテーションが付与されていないこと、あるいは **ejb-jar.xml** ファイルで EJB bean クラスとして宣言されていること。
- インターフェース **javax.enterprise.inject.spi.Extension** を実装しないこと。
- パラメーターのないコンストラクターか、**@Inject** アノテーションが付与されたコンストラクターがあること。
- @Vetoed** アノテーションが付いていないこと、または **@Vetoed** アノテーションが付けられたパッケージ内でないこと。

管理対象 bean の無制限の bean 型には、直接的あるいは間接的に実装する bean クラス、スーパークラスすべて、およびインターフェースすべてが含まれます。

管理対象 Bean にパブリックフィールドがある場合は、デフォルトの **@Dependent** スコープが必要です。

@Vetoed

@Vetoed アノテーションは CDI 1.1 で導入されました。このアノテーションを追加することにより、Bean をインジェクションから除外できます。

```
@Vetoed
public class SimpleGreeting implements Greeting {
    ...
}
```

このコードでは、**SimpleGreeting** Bean はインジェクションの対象となりません。

パッケージ内のすべての bean をインジェクションから除外できます。

```
@Vetoed
package org.sample.beans;

import javax.enterprise.inject.Vetoed;
```

org.sample.beans パッケージ内の **package-info.java** のこのコードにより、このパッケージ内のすべての Bean がインジェクションから除外されます。

ステートレス EJB や JAX-RS リソースエンドポイントなどの Java EE コンポーネントには、bean と見なされないように **@Vetoed** を付けることができます。 **@Vetoed** アノテーションをすべての永続エンティティに追加すると、**BeanManager** がエンティティを CDI Bean として管理しないようになります。エンティティに **@Vetoed** アノテーションが付けられた場合は、インジェクションが行われません。これは、JPA プロバイダーの破損原因となる操作を **BeanManager** が実行しないようにするためです。

7.4.2. CDI を用いたオブジェクトの Bean へのインジェクト

CDI コンポーネントがアプリケーションで検出されると、CDI は自動的にアクティベートされます。デフォルト値と異なるよう設定をカスタマイズする場合は、デプロイメントアーカイブに **META-INF/beans.xml** ファイルまたは **WEB-INF/beans.xml** ファイルを含めることができます。

他のオブジェクトにオブジェクトをインジェクトする

1. クラスのインスタンスを取得するには、bean 内でフィールドに **@Inject** アノテーションを付けます。

```
public class TranslateController {
    @Inject TextTranslator textTranslator;
    ...
}
```

2. インジェクトしたオブジェクトのメソッドを直接使用します。 **TextTranslator** にメソッド **translate** があることを前提とします。

```
// in TranslateController class

public void translate() {
    translation = textTranslator.translate(inputText);
}
```

- Bean のコンストラクターでインジェクションを使用します。ファクトリーやサービスロケータを使用して作成する代わりに、Bean のコンストラクターへオブジェクトをインジェクトできます。

```
public class TextTranslator {

    private SentenceParser sentenceParser;
    private Translator sentenceTranslator;

    @Inject
    TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
        this.sentenceParser = sentenceParser;
        this.sentenceTranslator = sentenceTranslator;
    }

    // Methods of the TextTranslator class
    ...
}
```

- Instance(<T>)** インターフェースを使用してインスタンスをプログラムにより取得します。Bean 型でパラメーター化されると、**Instance** インターフェースは **TextTranslator** のインスタンスを返すことができます。

```
@Inject Instance<TextTranslator> textTranslatorInstance;
...
public void translate() {
    textTranslatorInstance.get().translate(inputText);
}
```

オブジェクトを Bean にインジェクトすると、Bean は全オブジェクトのメソッドとプロパティを使用できるようになります。Bean のコンストラクターにインジェクトするときに、インジェクションがすでに存在するインスタンスを参照する場合以外は、Bean のコンストラクターが呼び出されるとインジェクトされたオブジェクトのインスタンスが作成されます。たとえば、セッションの存続期間内にセッションスコープの Bean をインジェクトしても、新しいインスタンスは作成されません。

7.5. コンテキストおよびスコープ

CDI では、特定のスコープに関連付けられた Bean のインスタンスを保持するストレージ領域をコンテキストと呼びます。

スコープは bean とコンテキスト間のリンクです。スコープとコンテキストの組み合わせは特定のライフサイクルを持つことができます。事前定義された複数のスコープが存在し、独自のスコープを作成できます。事前定義されたスコープの例は **@RequestScoped**、**@SessionScoped**、および **@ConversationScope** です。

表7.1 利用可能なスコープ

範囲	説明
@Dependent	Bean は、参照を保持する Bean のライフサイクルにバインドされます。インジェクション Bean のデフォルトのスコープは @Dependent です。
@ApplicationScoped	Bean はアプリケーションのライフサイクルにバインドされます。

範囲	説明
@RequestScoped	Bean はリクエストのライフサイクルにバインドされます。
@SessionScoped	Bean はセッションのライフサイクルにバインドされます。
@ConversationScoped	Bean は会話のライフサイクルにバインドされます。会話スコープは、リクエストの長さでセッションの間であり、アプリケーションによって制御されます。
カスタムスコープ	上記のコンテキストで対応できない場合は、カスタムスコープを定義できます。

7.6. 名前付き BEAN

Bean には、**@Named** アノテーションを使用して名前を付けることができます。Bean を命名することにより、Bean を Java Server Faces (JSF) と Expression Language (EL) で直接使用できるようになります。

@Named アノテーションは、bean 名である任意のパラメーターを取ります。このパラメーターが省略された場合、デフォルトの bean 名は、最初の文字が小文字に変換された bean のクラス名になります。

7.6.1. 名前付き Bean の使用

@Named Annotation を使用した Bean 名の設定

1. **@Named** アノテーションを使用して名前を Bean に割り当てます。

```
@Named("greeter")
public class GreeterBean {
    private Welcome welcome;

    @Inject
    void init (Welcome welcome) {
        this.welcome = welcome;
    }

    public void welcomeVisitors() {
        System.out.println(welcome.buildPhrase("San Francisco"));
    }
}
```

上記の例では、名前が指定されていない場合、デフォルトの名前は **greeterBean** になります。

2. JSF ビューで名前付き Bean を使用します。

```
<h:form>
    <h:commandButton value="Welcome visitors" action="#{greeter.welcomeVisitors}"/>
</h:form>
```

7.7. BEAN ライフサイクル

このタスクは、リクエストの残存期間の間 Bean を保存する方法を示しています。

インジェクション Bean のデフォルトのスコープは **@Dependent** です。つまり、bean のライフサイクルは、参照を保持する bean のライフサイクルに依存します。他の複数のスコープが存在し、独自のスコープを定義できます。詳細は、「[コンテキストおよびスコープ](#)」を参照してください。

Bean ライフサイクルの管理

1. 必要なスコープで bean にアノテーションを付けます。

```
@RequestScoped
@Named("greeter")
public class GreeterBean {
    private Welcome welcome;
    private String city; // getter & setter not shown
    @Inject void init(Welcome welcome) {
        this.welcome = welcome;
    }
    public void welcomeVisitors() {
        System.out.println(welcome.buildPhrase(city));
    }
}
```

2. Bean が JSF ビューで使用されると、Bean は状態を保持します。

```
<h:form>
  <h:inputText value="#{greeter.city}"/>
  <h:commandButton value="Welcome visitors" action="#{greeter.welcomeVisitors}"/>
</h:form>
```

Bean は、指定するスコープに関連するコンテキストに保存され、スコープが適用される限り存続します。

7.7.1. プロデューサーメソッドの使用

プロデューサーメソッドは、Bean インスタンスのソースとして動作するメソッドです。指定されたコンテキストにインスタンスが存在しない場合は、メソッド宣言自体で Bean が定義され、コンテナによって Bean のインスタンスを取得するメソッドが呼び出されます。プロデューサーメソッドにより、アプリケーションは Bean インスタンス化プロセスを完全に制御できるようになります。

ここでは、インジェクション用の bean ではないさまざまなオブジェクトを生成するプロデューサーメソッドを使用する方法を示します。

例: プロデューサーメソッドの使用

代替の代わりにプロデューサーメソッドを使用すると、デプロイメント後のポリモーフィズムが可能になります。

例の **@Preferred** アノテーションは、修飾子アノテーションです。修飾子の詳細は、「[修飾子](#)」を参照してください。

```
@SessionScoped
public class Preferences implements Serializable {
```

```

private PaymentStrategyType paymentStrategy;
...
@Produces @Preferred
public PaymentStrategy getPaymentStrategy() {
    switch (paymentStrategy) {
        case CREDIT_CARD: return new CreditCardPaymentStrategy();
        case CHECK: return new CheckPaymentStrategy();
        default: return null;
    }
}
}

```

以下のインジェクションポイントは、プロデューサーメソッドと同じタイプおよび修飾子アノテーションを持つため、通常の CDI インジェクションルールを使用してプロデューサーメソッドに解決されます。プロデューサーメソッドは、このインジェクションポイントを処理するインスタンスを取得するためにコンテナにより呼び出されます。

```
@Inject @Preferred PaymentStrategy paymentStrategy;
```

例: プロデューサーメソッドへのスコープの割り当て

プロデューサーメソッドのデフォルトのスコープは **@Dependent** です。スコープを Bean に割り当てた場合、スコープは適切なコンテキストにバインドされます。この例のプロデューサーメソッドは、1つのセッションあたり一度だけ呼び出されます。

```

@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy() {
    ...
}

```

例: プロデューサーメソッド内部でのインジェクションの使用

アプリケーションにより直接インスタンス化されたオブジェクトは、依存関係の注入を利用できず、インターセプターを持ちません。ただし、プロデューサーメソッドへの依存関係の注入を使用して Bean インスタンスを取得できます。

```

@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(CreditCardPaymentStrategy ccps,
                                           CheckPaymentStrategy cps ) {
    switch (paymentStrategy) {
        case CREDIT_CARD: return ccps;
        case CHEQUE: return cps;
        default: return null;
    }
}

```

リクエストスコープの Bean をセッションスコープのプロデューサーにインジェクトする場合は、プロデューサーメソッドにより、現在のリクエストスコープのインスタンスがセッションスコープにプロモートされます。これは、適切な動作ではないため、プロデューサーメソッドをこのように使用する場合は注意してください。



注記

プロデューサーメソッドのスコープは、プロデューサーメソッドを宣言する Bean から継承されません。

プロデューサーメソッドを使用して、Bean ではないオブジェクトをインジェクトし、コードを動的に変更できます。

7.8. 代替の BEAN

実装が特定のクライアントモジュールまたはデプロイメントシナリオに固有である Bean が代替となります。

デフォルトでは、**@Alternative** Bean が無効になります。これらは、**beans.xml** ファイルを編集することにより、特定の Bean アーカイブに対して有効になります。ただし、このアクティベーションは、そのアーカイブの Bean に対してのみ適用されます。CDI 1.1 以降、代替の Bean は、**@Priority** アノテーションを使用してアプリケーション全体に対して有効にできます。

例: 代替の定義

この代替により、**@Synchronous** 代替と **@Asynchronous** 代替を使用して **PaymentProcessor** クラスの実装が定義されます。

```
@Alternative @Synchronous @Asynchronous

public class MockPaymentProcessor implements PaymentProcessor {

    public void process(Payment payment) { ... }

}
```

例: beans.xml を使用した @Alternative の有効化

```
<beans
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/beans_2_0.xsd">
  <alternatives>
    <class>org.mycompany.mock.MockPaymentProcessor</class>
  </alternatives>
</beans>
```

選択された代替の宣言

@Priority アノテーションによって、アプリケーション全体に対して代替を有効にすることができます。代替にはアプリケーションの優先度を割り当てることができます。

- 管理対象 Bean またはセッション Bean の Bean クラスに **@Priority** アノテーションを置く、または
- プロデューサーメソッド、フィールド、またはリソースを宣言する Bean クラスに **@Priority** アノテーションを置く

7.8.1. 代替を用いたインジェクションのオーバーライド

代替の Bean を使用すると、既存の Bean をオーバーライドできます。これらは、同じ役割を満たすクラスをプラグインする方法として考慮できますが、動作が異なります。代替の Bean はデフォルトで無効になります。

このタスクは、代替を指定し、有効にする方法を示しています。

インジェクションのオーバーライド

このタスクでは、プロジェクトに **TranslatingWelcome** クラスがすでにあることを前提としています。ただし、これを "mock" **TranslatingWelcome** クラスでオーバーライドするとします。これは、実際の **Translator** Bean を使用できないテストデプロイメントのケースに該当します。

1. 代替を定義します。

```
@Alternative
@Translating
public class MockTranslatingWelcome extends Welcome {
    public String buildPhrase(string city) {
        return "Bienvenue Ã " + city + "!";
    }
}
```

2. 置換実装をアクティベートするために、完全修飾クラス名を **META-INF/beans.xml** または **WEB-INF/beans.xml** ファイルに追加します。

```
<beans>
  <alternatives>
    <class>com.acme.MockTranslatingWelcome</class>
  </alternatives>
</beans>
```

元の実装の代わりに代替実装が使用されます。

7.9. ステレオタイプ

多くのシステムでは、アーキテクチャパターンを使用して繰り返し発生する Bean ロールのセットを生成します。ステレオタイプを使用すると、このようなルールを指定し、中心的な場所で、このルールを持つ Bean に対する共通メタデータを宣言できます。

ステレオタイプにより、以下のいずれかの組み合わせがカプセル化されます。

- デフォルトのスコープ。
- インターセプターバインディングのセット。

また、ステレオタイプにより、以下の2つのいずれかを指定できます。

- ステレオタイプがデフォルトの bean EL 名であるすべての bean。
- ステレオタイプが代替であるすべての bean。

Bean は、ステレオタイプを 0 個以上宣言できます。ステレオタイプは、他の複数のアノテーションをパッケージ化する **@Stereotype** アノテーションです。ステレオタイプアノテーションは、bean クラス、プロデューサーメソッド、またはフィールドに適用できます。

ステレオタイプからスコープを継承するクラスは、そのステレオタイプをオーバーライドし、bean に直接スコープを指定できます。

また、ステレオタイプが **@Named** アノテーションを持つ場合、配置された bean はデフォルトの bean 名を持ちます。この bean は、**@Named** アノテーションが bean で直接指定された場合に、この名前をオーバーライドできます。名前付き bean の詳細は、「[名前付き Bean](#)」を参照してください。

7.9.1. ステレオタイプの使用

ステレオタイプを使用しないと、アノテーションが煩雑になる可能性があります。このタスクは、ステレオタイプを使用して煩雑さを軽減し、コードを効率化する方法を示しています。

例: 煩雑なアノテーション

```
@Secure
@Transactional
@RequestScoped
@Named
public class AccountManager {
    public boolean transfer(Account a, Account b) {
        ...
    }
}
```

ステレオタイプの定義および使用

1. ステレオタイプを定義します。

```
@Secure
@Transactional
@RequestScoped
@Named
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface BusinessComponent {
    ...
}
```

2. ステレオタイプを使用します。

```
@BusinessComponent
public class AccountManager {
    public boolean transfer(Account a, Account b) {
        ...
    }
}
```

7.10. オブザーバーメソッド

オブザーバーメソッドは、イベント発生時に通知を受け取ります。

また、CDI は、イベントが発生したトランザクションの完了前または完了後フェーズ中にイベント通知を受け取るトランザクションオブザーバーメソッドを提供します。

7.10.1. イベントの発生と確認

例: イベントの実行

以下のコードは、メソッドでインジェクトおよび使用されるイベントを示しています。

```
public class AccountManager {
    @Inject Event<Withdrawal> event;

    public boolean transfer(Account a, Account b) {
        ...
        event.fire(new Withdrawal(a));
    }
}
```

例: 修飾子を使用したイベントの実行

修飾子を使用すると、より具体的にイベントのインジェクションにアノテーションを付けられます。修飾子の詳細は、「[修飾子](#)」を参照してください。

```
public class AccountManager {
    @Inject @Suspicious Event <Withdrawal> event;

    public boolean transfer(Account a, Account b) {
        ...
        event.fire(new Withdrawal(a));
    }
}
```

例: イベントの確認

イベントを確認するには、**@Observes** アノテーションを使用します。

```
public class AccountObserver {
    void checkTran(@Observes Withdrawal w) {
        ...
    }
}
```

修飾子を使用すると、特定の種類のイベントのみを確認できます。

```
public class AccountObserver {
    void checkTran(@Observes @Suspicious Withdrawal w) {
        ...
    }
}
```

7.10.2. トランザクションオブザーバー

トランザクションオブザーバーは、イベントが発生したトランザクションの完了フェーズ前または完了フェーズ後にイベント通知を受け取ります。トランザクションオブザーバーは、単一のアトミックトランザクションよりも状態が保持される期間が長いため、トランザクションオブザーバーはステートフルオブジェクトモデルで重要になります。

トランザクションオブザーバーには5つの種類があります。

- **IN_PROGRESS**: デフォルトではオブザーバーは即座に呼び出されます。

- **AFTER_SUCCESS**: トランザクションが正常に完了する場合のみ、オブザーバーはトランザクションの完了フェーズの後に呼び出されます。
- **AFTER_FAILURE**: トランザクションの完了に失敗する場合のみ、オブザーバーはトランザクションの完了フェーズの後に呼び出されます。
- **AFTER_COMPLETION**: オブザーバーはトランザクションの完了フェーズの後に呼び出されません。
- **BEFORE_COMPLETION**: オブザーバーはトランザクションの完了フェーズの前に呼び出されます。

以下のオブザーバーメソッドは、カテゴリツリーを更新するトランザクションが正常に実行される場合のみアプリケーションコンテキストにキャッシュされたクエリー結果セットを更新します。

```
public void refreshCategoryTree(@Observes(during = AFTER_SUCCESS) CategoryUpdateEvent event) { ... }
```

以下の例のように、JAP クエリーの結果セットをアプリケーションスコープでキャッシュしたと仮定します。

```
import javax.ejb.Singleton;
import javax.enterprise.inject.Produces;

@ApplicationScoped @Singleton

public class Catalog {
    @PersistenceContext EntityManager em;
    List<Product> products;
    @Produces @Catalog
    List<Product> getCatalog() {
        if (products==null) {
            products = em.createQuery("select p from Product p where p.deleted = false")
                .getResultList();
        }
        return products;
    }
}
```

Product はときどき作成および削除されます。この場合は、**製品** カタログを更新する必要があります。ただし、この更新を実行する前に、トランザクションが正常に完了するのを待つ必要があります。

以下は、イベントを引き起こす **Products** を作成および削除する bean の例になります。

```
import javax.enterprise.event.Event;

@Stateless

public class ProductManager {
    @PersistenceContext EntityManager em;
    @Inject @Any Event<Product> productEvent;
    public void delete(Product product) {
        em.delete(product);
        productEvent.select(new AnnotationLiteral<Deleted>({})).fire(product);
    }
}
```

```

public void persist(Product product) {
    em.persist(product);
    productEvent.select(new AnnotationLiteral<Created>()).fire(product);
}
...
}

```

トランザクションが正常に完了した後に、**Catalog** がイベントを監視できるようになりました。

```

import javax.ejb.Singleton;

@ApplicationScoped @Singleton
public class Catalog {
    ...
    void addProduct(@Observes(during = AFTER_SUCCESS) @Created Product product) {
        products.add(product);
    }

    void removeProduct(@Observes(during = AFTER_SUCCESS) @Deleted Product product) {
        products.remove(product);
    }
}

```

7.11. インターセプター

インターセプターを使用すると、Bean のメソッドを直接変更せずに Bean のビジネスメソッドに機能を追加できます。インターセプターは、Bean のビジネスメソッドの前に実行されます。インターセプターは、[JSR 345: Enterprise JavaBeans™ 3.2](#) 仕様の一部として定義されています。

CDI により、インターセプターと Bean をバインドするアノテーションを利用できるため、この機能が強化されます。

インターセプションポイント

- **ビジネスメソッドインターセプション:** ビジネスメソッドのインターセプターは、Bean のクライアントによる Bean のメソッド呼び出しに適用されます。
- **ライフサイクルコールバックインターセプション:** ライフサイクルコールバックインターセプションは、コンテナーによるライフサイクルコールバックの呼び出しに適用されます。
- **タイムアウトメソッドインターセプター:** タイムアウトメソッドインターセプターは、コンテナーによる EJB タイムアウトメソッドの呼び出しに適用されます。

インターセプターの有効化

デフォルトでは、すべてのインターセプターが無効になります。インターセプターは、Bean アーカイブの **beans.xml** 記述子を使用して有効にすることができます。ただし、このアクティベーションは、そのアーカイブの Bean に対してのみ適用されます。CDI 1.1 以降、インターセプターは、**@Priority** アノテーションを使用してアプリケーション全体に対して有効にできます。

例: beans.xml のインターセプターの有効化

```

<beans
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"

```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
  http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/beans_2.0.xsd">
<interceptors>
  <class>org.mycompany.myapp.TransactionInterceptor</class>
</interceptors>
</beans>

```

XML 宣言を使用すると、以下の 2 つのことが可能になります。

- システムでインターセプターの順序を指定して、結果が正確になるようにすることができます。
- デプロイメント時にインターセプタークラスを有効または無効にすることができます。

@Priority を使用して有効にされたインターセプターは、**beans.xml** ファイルを使用して有効にされたインターセプターよりも前に呼び出されます。



注記

インターセプターが **@Priority** によって有効になり、同時に **beans.xml** ファイルによって呼び出されると、移植不可能な動作を引き起こします。したがって、複数の CDI 実装全体で整合性のある動作を維持するには、この組み合わせで有効にしないでください。

7.11.1. CDI とのインターセプターの使用

CDI により、インターセプターコードが単純化され、ビジネスコードへの適用が簡単になります。

CDI がない場合、インターセプターには 2 つの問題があります。

- Bean は、インターセプター実装を直接指定する必要があります。
- アプリケーションの各 Bean は、インターセプターの完全なセットを適切な順序で指定する必要があります。この場合、アプリケーション全体でインターセプターを追加または削除するには時間がかかり、エラーが発生する傾向があります。

CDI とインターセプターの使用

1. インターセプターバインディングタイプを定義します。

```

@InterceptorBinding
@Retention(RUNTIME)
@Target({TYPE, METHOD})
public @interface Secure {}

```

2. インターセプター実装をマーク付けします。

```

@Secure
@Interceptor
public class SecurityInterceptor {
  @AroundInvoke
  public Object aroundInvoke(InvocationContext ctx) throws Exception {
    // enforce security ...
  }
}

```

```

    return ctx.proceed();
  }
}

```

3. 開発環境でインターセプターを使用します。

```

@Secure
public class AccountManager {
    public boolean transfer(Account a, Account b) {
        ...
    }
}

```

4. **META-INF/beans.xml** または **WEB-INF/beans.xml** ファイルに追加して、デプロイメントでインターセプターを有効にします。

```

<beans>
  <interceptors>
    <class>com.acme.SecurityInterceptor</class>
    <class>com.acme.TransactionInterceptor</class>
  </interceptors>
</beans>

```

インターセプターは、リストされた順序で適用されます。

7.12. デコレーター

デコレーターは、特定の Java インターフェースからの呼び出しをインターセプトし、そのインターフェースに割り当てられたすべてのセマンティクスを認識します。デコレーターは、何らかの業務をモデル化するのに役に立ちますが、インターセプターの一般性を持ちません。デコレーターは Bean または抽象クラスであり、デコレートするタイプを実装し、**@Decorator** アノテーションが付けられます。CDI アプリケーションでデコレーターを呼び出すには、**beans.xml** ファイルで指定する必要があります。

例: beans.xml でのデコレーターの呼び出し

```

<beans
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/beans_2_0.xsd">
  <decorators>
    <class>org.mycompany.myapp.LargeTransactionDecorator</class>
  </decorators>
</beans>

```

この宣言には 2 つの主な目的があります。

- システムでデコレーターの順序を指定して、結果が正確になるようにすることができます。
- デプロイメント時にデコレータークラスを有効または無効にすることができます。

デコレートされたオブジェクトへの参照を取得するために、デコレーターには **@Delegate** インジェクションポイントが1つ必要になります。

例: デコレータークラス

```
@Decorator
public abstract class LargeTransactionDecorator implements Account {

    @Inject @Delegate @Any Account account;
    @PersistenceContext EntityManager em;

    public void withdraw(BigDecimal amount) {
        ...
    }

    public void deposit(BigDecimal amount);
    ...
}
}
```

CDI 1.1以降、デコレーターは、**@Priority** アノテーションを使用してアプリケーション全体に対して有効にできます。

@Priority を使用して有効になったデコレーターは、**beans.xml** ファイルを使用して有効になったデコレーターよりも前に呼び出されます。優先度の値が小さいものが最初に呼び出されます。



注記

デコレーターが **@Priority** によって有効になり、同時に **beans.xml** によって呼び出されると、移移植不可能な動作を引き起こします。したがって、複数の CDI 実装全体で整合性のある動作を維持するには、この組み合わせで有効にしないでください。

7.13. 移植可能な拡張機能

CDI は、フレームワーク、拡張機能、および他のテクノロジーとの統合の基礎となることを目的としています。したがって、CDI は、移植可能な CDI の拡張機能の開発者が使用する SPI のセットを公開します。

拡張機能は、以下のような種類の機能を提供できます。

- ビジネスプロセス管理エンジンとの統合。
- Spring、Seam、GWT、Wicket などのサードパーティーフレームワークとの統合。
- CDI プログラミングモデルに基づく新しいテクノロジー。

JSR-346 仕様に基づいて、移植可能な拡張機能は次の方法でコンテナと統合できます。

- 独自の Bean、インターセプター、およびデコレーターをコンテナに提供します。
- 依存関係注入サービスを使用した独自のオブジェクトへの依存関係のインジェクション。
- カスタムスコープのコンテキスト実装を提供します。

- アノテーションベースのメタデータを別のソースからのメタデータで拡大またはオーバーライドします。

詳細は、Weld ドキュメントの「[Portable extensions](#)」を参照してください。

7.14. BEAN プロキシ

通常、インジェクトされた bean のクライアントは bean インスタンスへの直接参照を保持しません。bean が依存オブジェクト (スコープ **@Dependent**) でない場合、コンテナーはプロキシオブジェクトを使用して、インジェクトされたすべての参照を bean にリダイレクトする必要があります。

この bean プロキシはクライアントプロキシと呼ばれ、メソッド呼び出しを受け取る bean インスタンスが、現在のコンテキストと関連するインスタンスになるようにします。またクライアントプロキシは、他のインジェクトされた bean を再帰的にシリアライズせずに、セッションコンテキストなどのコンテキストにバインドされる bean をディスクヘシリアライズできるようにします。

Java の制限により、コンテナーによるプロキシの作成が不可能な Java の型があります。これらの型の1つで宣言されたインジェクションポイントが、**@Dependent** 以外のスコープを持つ bean に解決すると、コンテナーがデプロイメントをアボートします。

特定の Java の型ではコンテナーによってプロキシを作成できません。これらの型には次のようなものがあります。

- パラメーターのない非プライベートコンストラクターを持たないクラス
- **final** が宣言されたクラスまたは **final** メソッドを持つクラス。
- アレイおよびプリミティブ型。

7.15. インジェクションでのプロキシの使用

各 bean のライフサイクルが異なる場合に、インジェクションにプロキシが使用されます。プロキシは起動時に作成された bean のサブクラスで、bean クラスのプライベートメソッド以外のメソッドをすべて上書きします。プロキシは実際の bean インスタンスへ呼び出しを転送します。

この例では、**PaymentProcessor** インスタンスは直接 **Shop** へインジェクトされません。その代わりにプロキシがインジェクトされ、**processPayment()** メソッドが呼び出されるとプロキシが現在の **PaymentProcessor** Bean インスタンスをルックアップし、**processPayment()** メソッドを呼び出します。

例: プロキシインジェクション

```
@ConversationScoped
class PaymentProcessor
{
    public void processPayment(int amount)
    {
        System.out.println("I'm taking $" + amount);
    }
}

@ApplicationScoped
public class Shop
{
```

```
@Inject  
PaymentProcessor paymentProcessor;  
  
public void buyStuff()  
{  
    paymentProcessor.processPayment(100);  
}  
}
```

第8章 JBOSS EAP MBEAN サービス

管理対象 Bean (単に MBean と呼ばれることもあります) は、依存関係インジェクションで作成された JavaBean の型です。MBean サービスは JBoss EAP サーバーの中心的な要素です。

8.1. JBOSS MBEAN SERVICE の記述

JBoss サービスに依存するカスタム MBean サービスを記述するには、サービスインターフェースメソッドパターンが必要です。JBoss MBean のサービスインターフェースメソッドパターンは **create**、**start**、**stop**、および **destroy** が実行可能である場合に MBean サービスに通知する複数のライフサイクル操作で構成されます。

以下の方法を使用すると依存関係の状態を管理できます。

- MBean で特定のメソッドを呼び出したい場合は、これらのメソッドを MBean インターフェースで宣言します。この方法では、MBean 実装で JBoss 固有クラスの依存関係を回避できません。
- JBoss 固有クラスの依存関係を気にしない場合は、MBean インターフェースで **ServiceMBean** インターフェースおよび **ServiceMBeanSupport** クラスを拡張できます。**ServiceMBeanSupport** クラスは create、start、および stop などのサービスライフサイクルメソッドの実装を提供します。**start()** イベントなどの特定のイベントを処理するには、**ServiceMBeanSupport** クラスによって提供される **startService()** メソッドをオーバーライドする必要があります。

8.1.1. 標準の MBean の例

ここでは、サービスアーカイブ (.sar) で一緒にパッケージ化される 2 つの MBean サービスの例を開発します。

ConfigServiceMBean インターフェースは **start**、**getTimeout**、および **stop** などの特定のメソッドを宣言し、JBoss 固有のクラスを使用せずに MBean に対して **start**、**hold**、および **stop** を実行します。**ConfigService** クラスは **ConfigServiceMBean** インターフェースを実装した後、このインターフェース内で使用されたメソッドを実装します。

PlainThread クラスは **ServiceMBeanSupport** クラスを拡張し、**PlainThreadMBean** インターフェースを実装します。**PlainThread** はスレッドを開始し、**ConfigServiceMBean.getTimeout()** を使用してスレッドのスリープ時間を決定します。

例: MBean サービスクラス

```
package org.jboss.example.mbean.support;
public interface ConfigServiceMBean {
    int getTimeout();
    void start();
    void stop();
}
package org.jboss.example.mbean.support;
public class ConfigService implements ConfigServiceMBean {
    int timeout;
    @Override
    public int getTimeout() {
        return timeout;
    }
    @Override
```

```

public void start() {
    //Create a random number between 3000 and 6000 milliseconds
    timeout = (int)Math.round(Math.random() * 3000) + 3000;
    System.out.println("Random timeout set to " + timeout + " seconds");
}
@Override
public void stop() {
    timeout = 0;
}
}

package org.jboss.example.mbean.support;
import org.jboss.system.ServiceMBean;
public interface PlainThreadMBean extends ServiceMBean {
    void setConfigService(ConfigServiceMBean configServiceMBean);
}

package org.jboss.example.mbean.support;
import org.jboss.system.ServiceMBeanSupport;
public class PlainThread extends ServiceMBeanSupport implements PlainThreadMBean {
    private ConfigServiceMBean configService;
    private Thread thread;
    private volatile boolean done;
    @Override
    public void setConfigService(ConfigServiceMBean configService) {
        this.configService = configService;
    }
    @Override
    protected void startService() throws Exception {
        System.out.println("Starting Plain Thread MBean");
        done = false;
        thread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    while (!done) {
                        System.out.println("Sleeping...");
                        Thread.sleep(configService.getTimeout());
                        System.out.println("Slept!");
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
        thread.start();
    }
    @Override
    protected void stopService() throws Exception {
        System.out.println("Stopping Plain Thread MBean");
        done = true;
    }
}

```

jboss-service.xml 記述子は、**inject** タグを使用して **ConfigService** クラスが **PlainThread** クラスにインジェクトされる方法を示します。**inject** タグは **PlainThreadMBean** と **ConfigServiceMBean** 間の依存関係を確立し、**PlainThreadMBean** が簡単に **ConfigServiceMBean** を使用できるようにします。

例: JBoss-service.xml サービス記述子

```
<server xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:jboss:service:7.0 jboss-service_7_0.xsd"
        xmlns="urn:jboss:service:7.0">
  <mbean code="org.jboss.example.mbean.support.ConfigService"
        name="jboss.support:name=ConfigBean"/>
  <mbean code="org.jboss.example.mbean.support.PlainThread"
        name="jboss.support:name=ThreadBean">
    <attribute name="configService">
      <inject bean="jboss.support:name=ConfigBean"/>
    </attribute>
  </mbean>
</server>
```

MBean の例を記述した後、クラスと **jboss-service.xml** 記述子をサービスアーカイブ (**.sar**) の **META-INF/** フォルダでパッケージ化できます。

8.2. JBOSS MBEAN サービスのデプロイ

例: 管理対象ドメインでの MBean のデプロイおよびテスト

以下のコマンドを使用して、MBean の例 (**ServiceMBeanTest.sar**) を管理対象ドメインでデプロイします。

```
deploy ~/Desktop/ServiceMBeanTest.sar --all-server-groups
```

例: スタンドアロンサーバーでの MBean のデプロイおよびテスト

以下のコマンドを使用して、スタンドアロンサーバーで MBean の例 (**ServiceMBeanTest.sar**) をビルドおよびデプロイします。

```
deploy ~/Desktop/ServiceMBeanTest.sar
```

例: MBean アーカイブのアンデプロイ

以下のコマンドを使用して、MBean の例をアンデプロイします。

```
undeploy ServiceMBeanTest.sar
```

第9章 コンカレンシーユーティリティー

コンカレンシーユーティリティーは、Java SE Concurrency Utilities を Java EE アプリケーション環境仕様で使用できるようにする API です。これは、「[JSR 236: Concurrency Utilities for Java™ EE](#)」で定義されています。JBoss EAP では、EE コンカレンシーユーティリティーのインスタンスの作成、編集、および削除を行えます。そのため、使用するアプリケーションにインスタンスをすぐに利用できるようにできます。

コンカレンシーユーティリティーを使用すると、既存のコンテキストのアプリケーションスレッドをプルし、独自のスレッドで使用するにより、呼び出しコンテキストを拡張できるようになります。呼び出しコンテキストのこの拡張には、デフォルトでクラスローディング、JNDI、およびセキュリティーコンテキストが含まれます。

コンカレンシーユーティリティーのタイプには以下のものが含まれます。

- コンテキストサービス
- 管理対象スレッドファクトリー
- 管理対象エグゼキューターサービス
- 管理対象スケジュール済みエグゼキューターサービス

例: `standalone.xml` のコンカレンシーユーティリティー

```
<subsystem xmlns="urn:jboss:domain:ee:4.0">
  <spec-descriptor-property-replacement>false</spec-descriptor-property-replacement>
  <concurrent>
    <context-services>
      <context-service name="default" jndi-name="java:jboss/ee/concurrency/context/default"
use-transaction-setup-provider="true"/>
    </context-services>
    <managed-thread-factories>
      <managed-thread-factory name="default" jndi-
name="java:jboss/ee/concurrency/factory/default" context-service="default"/>
    </managed-thread-factories>
    <managed-executor-services>
      <managed-executor-service name="default" jndi-
name="java:jboss/ee/concurrency/executor/default" context-service="default" hung-task-
threshold="60000" keepalive-time="5000"/>
    </managed-executor-services>
    <managed-scheduled-executor-services>
      <managed-scheduled-executor-service name="default" jndi-
name="java:jboss/ee/concurrency/scheduler/default" context-service="default" hung-task-
threshold="60000" keepalive-time="3000"/>
    </managed-scheduled-executor-services>
  </concurrent>
  <default-bindings context-service="java:jboss/ee/concurrency/context/default"
datasource="java:jboss/datasources/ExampleDS" managed-executor-
service="java:jboss/ee/concurrency/executor/default" managed-scheduled-executor-
service="java:jboss/ee/concurrency/scheduler/default" managed-thread-
factory="java:jboss/ee/concurrency/factory/default"/>
</subsystem>
```

9.1. コンテキストサービス

コンテキストサービス (`javax.enterprise.concurrent.ContextService`) を使用すると、既存のオブジェクトからコンテキストプロキシをビルドできます。コンテキストプロキシにより、コンテキストが作成または呼び出されたとき (呼び出しが元のオブジェクトに転送される前) に他のコンカレンシーユーティリティによって使用される呼び出しコンテキストが準備されます。

コンテキストサービスコンカレンシーユーティリティの属性には以下のものが含まれます。

- **name:** すべてのコンテキストサービス内の一意の名前。
- **jndi-name:** JNDI でコンテキストサービスを配置する場所を定義します。
- **use-transaction-setup-provider:** 任意。プロキシオブジェクトを呼び出す場合に、コンテキストサービスによってビルドされたコンテキストプロキシがコンテキストでトランザクションを一時停止するかどうかを示します。デフォルト値は **false** ですが、デフォルトのコンテキストサービスの値は **true** です。

コンテキストサービスコンカレンシーユーティリティの使用方法は、上記の例を参照してください。

例: 新しいコンテキストサービスの追加

```
/subsystem=ee/context-service=newContextService:add(jndi-name=java:jboss/ee/concurrency/contextservice/newContextService)
```

例: コンテキストサービスの変更

```
/subsystem=ee/context-service=newContextService:write-attribute(name=jndi-name,value=java:jboss/ee/concurrency/contextservice/changedContextService)
```

この操作にはリロードが必要です。

例: コンテキストサービスの削除

```
/subsystem=ee/context-service=newContextService:remove()
```

この操作にはリロードが必要です。

9.2. 管理対象スレッドファクトリー

管理対象スレッドファクトリー (`javax.enterprise.concurrent.ManagedThreadFactory`) コンカレンシーユーティリティを使用すると、Java EE アプリで Java スレッドを作成できます。JBoss EAP は管理対象スレッドファクトリーインスタンスを処理するため、Java EE アプリケーションはライフサイクル関連メソッドを呼び出すことができません。

管理対象スレッドファクトリーコンカレンシーユーティリティの属性には以下のものがあります。

- **context-service:** すべての管理対象スレッドファクトリー内の一意の名前。
- **jndi-name:** JNDI で管理対象スレッドファクトリーを配置する場所を定義します。
- **priority:** 任意。ファクトリーによって作成された新しいスレッドの優先度を示します。デフォルトは **5** です。

例: 新しい管理対象スレッドファクトリーの追加

```
/subsystem=ee/managed-thread-factory=newManagedTF:add(context-service=newContextService,
jndi-name=java:jboss/ee/concurrency/threadfactory/newManagedTF, priority=2)
```

例: 管理対象スレッドファクトリーの変更

```
/subsystem=ee/managed-thread-factory=newManagedTF:write-attribute(name=jndi-name,
value=java:jboss/ee/concurrency/threadfactory/changedManagedTF)
```

この操作にはリロードが必要です。同様に、他の属性を変更することもできます。

例: 管理対象スレッドファクトリーの削除

```
/subsystem=ee/managed-thread-factory=newManagedTF:remove()
```

この操作にはリロードが必要です。

9.3. 管理対象エグゼキューターサービス

管理対象エグゼキューターサービス (**javax.enterprise.concurrent.ManagedExecutorService**) を使用すると、Java EE アプリケーションで非同期実行向けタスクを送信できます。JBoss EAP は管理対象エグゼキューターサービスインスタンスを処理するため、Java EE アプリケーションはライフサイクル関連メソッドを呼び出すことができません。

管理対象エグゼキューターサービスコンカレンシーユーティリティーの属性には以下のものがあります。

- **context-service**: オプション。既存のコンテキストサービスをその名前で参照します。指定された場合は、参照されたコンテキストサービスがタスクをエグゼキューターに送信したときに存在する呼び出しコンテキストを取得します (このコンテキストはタスクの実行時に使用されます)。
- **jndi-name**: JNDI で管理対象スレッドファクトリーを配置する場所を定義します。
- **max-threads**: エグゼキューターによって使用されるスレッドの最大数を定義します。未定義の場合は、**core-threads** からの値が使用されます。
- **thread-factory**: 既存の管理対象スレッドファクトリーをその名前で参照して内部スレッドの作成を処理します。指定されない場合は、デフォルト設定の管理対象スレッドファクトリーが作成され、内部で使用されます。
- **core-threads**: エグゼキューターによって使用されるスレッドの最小数。この属性を定義しないと、プロセッサの数を基にしてデフォルトが算出されます。**0** を値として指定することは推奨されません。キューイングストラテジーの決定にこの値がどのように使用されるかは、**queue-length** 属性を参照してください。
- **keepalive-time**: 内部スレッドをアイドル状態にできる時間 (ミリ秒単位) を定義します。属性のデフォルト値は **60000** です。
- **queue-length**: エグゼキューターのタスクキューの容量を示します。長さが **0** の場合は直接ハンドオフを意味し、拒否される可能性があります。この属性が定義されていない場合または **Integer.MAX_VALUE** に設定された場合は、非有界のキューが使用されるべきであることを示します。他のすべての値は正確なキューのサイズを指定します。非有界のキューまたは直接ハンドオフが使用される場合は、**0** よりも大きな **core-threads** の値が必要になります。

- **hung-task-threshold**: ミリ秒単位の時間を定義します。この時間が経過すると、管理対象エグゼキューターサービスによってタスクがハング状態にあると見なされ、強制終了します。値が 0 (デフォルト値) の場合、タスクはハング状態にあると見なされません。
- **long-running-tasks**: 長時間実行されるタスクの実行の最適化を推奨します。デフォルト値は **false** です。
- **reject-policy**: タスクがエグゼキューターによって拒否されたときに使用するポリシーを定義します。属性値は、デフォルト値で、例外が発生する **ABORT**、または例外が発生する前にエグゼキューターがもう 1 度送信を試みる **RETRY_ABORT** のいずれかになります。

例: 新しい管理対象エグゼキューターサービスの追加

```
/subsystem=ee/managed-executor-service=newManagedExecutorService:add(jndi-name=java:boss/ee/concurrency/executor/newManagedExecutorService, core-threads=7, thread-factory=default)
```

例: 管理対象エグゼキューターサービスの変更

```
/subsystem=ee/managed-executor-service=newManagedExecutorService:write-attribute(name=core-threads,value=10)
```

この操作にはリロードが必要です。同様に、他の属性を変更することもできます。

例: 管理対象エグゼキューターサービスの削除

```
/subsystem=ee/managed-executor-service=newManagedExecutorService:remove()
```

この操作にはリロードが必要です。

9.4. 管理対象スケジュール済みエグゼキューターサービス

管理対象スケジュール済みエグゼキューターサービス

(**javax.enterprise.concurrent.ManagedScheduledExecutorService**) を使用すると、Java EE アプリケーションで非同期実行向けタスクをスケジュールできます。JBoss EAP は管理対象スケジュール済みエグゼキューターサービスインスタンスを処理するため、Java EE アプリケーションはライフサイクル関連メソッドを呼び出すことができません。

管理対象エグゼキューターサービスコンカレンシーユーティリティの属性には以下のものがあります。

- **context-service**: 既存のコンテキストサービスをその名前参照します。指定された場合は、参照されたコンテキストサービスがタスクをエグゼキューターに送信したときに存在する呼び出しコンテキストを取得します (このコンテキストはタスクの実行時に使用されます)。
- **hung-task-threshold**: ミリ秒単位の時間を定義します。この時間が経過すると、管理対象のスケジュール設定されたエグゼキューターサービスによってタスクがハング状態にあると見なされ、強制終了します。値が 0 (デフォルト値) の場合、タスクはハング状態にあると見なされません。
- **keepalive-time**: 内部スレッドをアイドル状態にできる時間 (ミリ秒単位) を定義します。属性のデフォルト値は **60000** です。

- **reject-policy**: タスクがエグゼキューターによって拒否されたときに使用するポリシーを定義します。属性値は、デフォルト値で、例外が発生する **ABORT**、または例外が発生する前にエグゼキューターがもう1度送信を試みる **RETRY_ABORT** のいずれかになります。
- **core-threads**: スケジュール済みエグゼキューターによって使用されるスレッドの最小数を定義します。
- **jndi-name**: JNDI で管理対象スケジュール済みエグゼキューターサービスを配置する場所を定義します。
- **long-running-tasks**: 長時間実行中のタスクの実行の最適化を推奨します。デフォルト値は `false` です。
- **thread-factory**: 既存の管理対象スレッドファクトリーをその名前で参照して内部スレッドの作成を処理します。指定されない場合は、デフォルト設定の管理対象スレッドファクトリーが作成され、内部で使用されます。

例: 新しい管理対象スケジュール済みエグゼキューターサービスの追加

```
/subsystem=ee/managed-scheduled-executor-  
service=newManagedScheduledExecutorService:add(jndi-  
name=java:jboss/ee/concurrency/scheduledexecutor/newManagedScheduledExecutorService, core-  
threads=7, context-service=default)
```

この操作にはリロードが必要です。

例: 管理対象スケジュール済みエグゼキューターサービスの変更

```
/subsystem=ee/managed-scheduled-executor-  
service=newManagedScheduledExecutorService:write-attribute(name=core-threads, value=10)
```

この操作にはリロードが必要です。同様に、他の属性を変更することができます。

例: 管理対象スケジュール済みエグゼキューターサービスの削除

```
/subsystem=ee/managed-scheduled-executor-  
service=newManagedScheduledExecutorService:remove()
```

この操作にはリロードが必要です。

第10章 UNDERTOW

10.1. UNDERTOW ハンドラーについて

Undertow は、ブロックタスクと非ブロックタスクの両方に使用するよう設計された Web サーバーです。JBoss EAP 7 では JBoss Web は Undertow に置き換わります。主な機能の一部は以下のとおりです。

- ハイパフォーマンス
- 組み込み可能
- Servlet 4.0
- Web ソケット
- リバースプロキシ

リクエストライフサイクル

クライアントがサーバーに接続するときに、Undertow によって `io.undertow.server.HttpServerConnection` が作成されます。クライアントがリクエストを送信するときに、リクエストは Undertow パーサーによって解析され、生成される `io.undertow.server.HttpServerExchange` はルートハンドラーに渡されます。ルートハンドラーが完了すると、以下の 4 つのいずれかのことが起こります。

- 交換が完了します。
リクエストチャンネルと応答チャンネルが完全に読み取られたり、書き込まれた場合に、交換が完了したと見なされます。リクエスト側は、GET や HEAD などのコンテンツがないリクエストの場合に、自動的に完全に読み取られたと見なされます。読み取り側は、ハンドラーが完全な応答を書き込み、応答チャンネルを閉じ、応答チャンネルを完全にフラッシュしたときに、完了したと見なされます。交換がすでに完了した場合は、どんなアクションも行われません。
- 交換を完了せずにルートハンドラーが通常どおり返されます。
この場合、交換は `HttpServerExchange.endExchange()` を呼び出して完了します。
- ルートハンドラーが例外で返されます。
この場合、**500** の応答コードが設定され、`HttpServerExchange.endExchange()` を使用して交換が終了します。
- ルートハンドラーは、`HttpServerExchange.dispatch()` が呼び出された後、または非同期 IO が開始された後に返すことができます。
この場合、ディスパッチされたタスクはディスパッチエグゼキューターに送信されます。また、非同期 IO がリクエストチャンネルまたは応答チャンネルのいずれかで開始された場合は、このタスクが開始されます。交換はどちらの場合でも完了しません。非同期タスクによって、処理が完了したときに交換が完了します。

`HttpServerExchange.dispatch()` の最も一般的な使用法は、実行をブロッキングが許可されない IO スレッドから、ブロッキング操作を許可するワーカースレッドに移動することです。

例: ワーカースレッドへのディスパッチ

```
public void handleRequest(final HttpServerExchange exchange) throws Exception {  
    if (exchange.isInIoThread()) {  
        exchange.dispatch(this);  
        return;  
    }  
}
```

```

    }
    //handler code
}

```

交換は呼び出しスタックが返されるまで実際にはディスパッチされないため、交換で一度に複数のスレッドがアクティブにならないようにすることができます。交換はスレッドセーフではありません。ただし、交換は、両方のスレッドが1度に変更しようとしないう限り、複数のスレッド間で渡すことができます。

交換の終了

交換を終了するには、リクエストチャンネルを読み取り、応答チャンネルで `shutdownWrites()` を呼び出し、フラッシュする方法と `HttpServerExchange.endExchange()` を呼び出す方法の2つがあります。`endExchange()` が呼び出された場合、Undertow はコンテンツが生成されたかどうかを確認します。生成された場合、Undertow はリクエストチャンネルを単にドレインし、応答チャンネルを閉じ、フラッシュします。生成されず、交換で登録されたデフォルトの応答リスナーがある場合は、Undertow によってそれらの各応答リスナーがデフォルトの応答を生成できるようになります。このメカニズムにより、デフォルトのエラーページが生成されます。

Undertow の設定に関する詳細は、JBoss EAP 『[設定ガイド](#)』の「[Web サーバーの設定](#)」を参照してください。

10.2. デプロイメントでの既存の UNDERTOW ハンドラーの使用

Undertow は、JBoss EAP にデプロイされたすべてのアプリケーションと使用できる、デフォルトのハンドラーセットを提供します。

デプロイメントでハンドラーを使用するには、`WEB-INF/undertow-handlers.conf` ファイルを追加する必要があります。

例: WEB-INF/undertow-handlers.conf ファイル

```
allowed-methods(methods='GET')
```

また、特定のケースで指定のハンドラーを提供するために、すべてのハンドラーは任意の述語を取ることできます。

例: 任意の述語がある WEB-INF/undertow-handlers.conf ファイル

```
path('/my-path') -> allowed-methods(methods='GET')
```

上記の例では、`allowed-methods` ハンドラーのみがパス `/my-path` に適用されます。

Undertow ハンドラーのデフォルトパラメーター

一部のハンドラーにはデフォルトのパラメーターがあり、名前を使用せずにハンドラー定義でそのパラメーターの値を指定できます。

例: デフォルトのパラメーターを使用する WEB-INF/undertow-handlers.conf ファイル

```
path('/a') -> redirect('/b')
```

また、`WEB-INF/jboss-web.xml` ファイルを更新して1つまたは複数のハンドラーの定義を含めることもできますが、`WEB-INF/undertow-handlers.conf` を使用することが推奨されます。

例: WEB-INF/jboss-web.xml ファイル

```

<jboss-web>
  <http-handler>
    <class-name>io.undertow.server.handlers.AllowedMethodsHandler</class-name>
    <param>
      <param-name>methods</param-name>
      <param-value>GET</param-value>
    </param>
  </http-handler>
</jboss-web>

```

提供される Undertow ハンドラーの完全リストは「[提供される Undertow ハンドラー](#)」を参照してください。

10.3. カスタムハンドラーの作成

カスタムハンドラーを定義する方法は2つあります。

1. [WEB-INF/jboss-web.xml ファイルの使用](#)
2. [WEB-INF/undertow-handlers.conf での定義](#)

WEB-INF/jboss-web.xml ファイルを使用したカスタムハンドラーの定義

カスタムハンドラーは **WEB-INF/jboss-web.xml** ファイルで定義できます。

例: WEB-INF/jboss-web.xml でのカスタマーハンドラーの定義

```

<jboss-web>
  <http-handler>
    <class-name>org.jboss.example.MyHttpHandler</class-name>
  </http-handler>
</jboss-web>

```

例: `HttpHandler` クラス

```

package org.jboss.example;

import io.undertow.server.HttpHandler;
import io.undertow.server.HttpServerExchange;

public class MyHttpHandler implements HttpHandler {
    private HttpHandler next;

    public MyHttpHandler(HttpHandler next) {
        this.next = next;
    }

    public void handleRequest(HttpServerExchange exchange) throws Exception {
        // do something
        next.handleRequest(exchange);
    }
}

```

WEB-INF/jboss-web.xml ファイルを使用して、カスタムハンドラーにパラメーターを設定することもできます。

例: WEB-INF/jboss-web.xml でのパラメーターの定義

```
<jboss-web>
  <http-handler>
    <class-name>org.jboss.example.MyHttpHandler</class-name>
    <param>
      <param-name>myParam</param-name>
      <param-value>foobar</param-value>
    </param>
  </http-handler>
</jboss-web>
```

これらのパラメーターが機能するには、ハンドラークラスに対応するセッターが必要です。

例: ハンドラーでのセッターメソッドの定義

```
package org.jboss.example;

import io.undertow.server.HttpHandler;
import io.undertow.server.HttpServerExchange;

public class MyHttpHandler implements HttpHandler {
    private HttpHandler next;
    private String myParam;

    public MyHttpHandler(HttpHandler next) {
        this.next = next;
    }

    public void setMyParam(String myParam) {
        this.myParam = myParam;
    }

    public void handleRequest(HttpServerExchange exchange) throws Exception {
        // do something, use myParam
        next.handleRequest(exchange);
    }
}
```

WEB-INF/undertow-handlers.conf ファイルでのカスタムハンドラーの定義

ハンドラーの定義に **WEB-INF/jboss-web.xml** を使用する代わりに、ハンドラーは **WEB-INF/undertow-handlers.conf** ファイルで定義することもできます。

```
myHttpHandler(myParam='foobar')
```

WEB-INF/undertow-handlers.conf で定義されたハンドラーが機能するには、以下の2つのものを作成する必要があります。

1. **HandlerWrapper** にラップされた **HandlerBuilder** (**undertow-handlers.conf** 向けの対応する構文を定義し、**HttpHandler** を作成します)。

例: HandlerBuilder クラス

```
package org.jboss.example;
```

```

import io.undertow.server.HandlerWrapper;
import io.undertow.server.HttpHandler;
import io.undertow.server.handlers.builder.HandlerBuilder;

import java.util.Collections;
import java.util.Map;
import java.util.Set;

public class MyHandlerBuilder implements HandlerBuilder {
    public String name() {
        return "myHttpHandler";
    }

    public Map<String, Class<?>> parameters() {
        return Collections.<String, Class<?>>singletonMap("myParam", String.class);
    }

    public Set<String> requiredParameters() {
        return Collections.emptySet();
    }

    public String defaultParameter() {
        return null;
    }

    public HandlerWrapper build(final Map<String, Object> config) {
        return new HandlerWrapper() {
            public HttpHandler wrap(HttpHandler handler) {
                MyHttpHandler result = new MyHttpHandler(handler);
                result.setMyParam((String) config.get("myParam"));
                return result;
            }
        };
    }
}

```

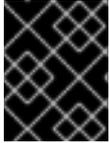
2. ファイルのエントリー。META-

INF/services/io.undertow.server.handlers.builder.HandlerBuilder。このファイルはクラスパス上である必要があります (例: **WEB-INF/classes**)。

```
org.jboss.example.MyHandlerBuilder
```

10.4. カスタム HTTP メカニズムの開発

Elytron を使用して Web アプリケーションをセキュアにする場合、**elytron** サブシステムを使用して登録できるカスタム HTTP 認証メカニズムを実装することが可能です。また、このメカニズムを利用するために、デプロイメントの変更を必要とせずにデプロイメント内の設定をオーバーライドすることも可能です。



重要

すべてのカスタム HTTP メカニズムは、**HttpServerAuthenticationMechanism** インターフェースを実装する必要があります。

通常、HTTP メカニズムでは、**HTTPServerRequest** オブジェクトを渡すリクエストの処理に **evaluateRequest** メソッドが呼び出されます。このメカニズムがリクエストを処理し、リクエスト上で以下のコールバックメソッドの1つを使用して、結果を示します。

- **authenticationComplete** - メカニズムによってリクエストが正常に認証されたことを示します。
- **authenticationFailed** - 認証は実行され、失敗したことを示します。
- **authenticationInProgress** - 認証は開始され、追加のラウンドトリップが必要であることを示します。
- **badRequest** - このメカニズムの認証によってリクエストの検証に失敗したことを示します。
- **noAuthenticationInProgress** - メカニズムが認証を何も実行しなかったことを示します。

HttpServerAuthenticationMechanism インターフェースを実行するカスタム HTTP メカニズムを作成したら、次にこのメカニズムのインスタンスを返すファクトリーを作成します。このファクトリーは、**HttpAuthenticationFactory** インターフェースを実装する必要があります。ファクトリーの実装で最も重要なのは、要求されたメカニズムの名前を二重チェックすることです。必要なメカニズムを作成できない場合は、ファクトリーが null を返すことが重要になります。メカニズムファクトリーは、要求されたメカニズムの作成が可能であるかどうかを決定するために、渡されたマップのプロパティーも考慮することができます。

メカニズムファクトリーが利用可能であるかどうかをアドバタイズするのに使用できる方法は2つあります。

- 1つ目は、サポートする各メカニズムに対して1度、利用可能なサービスとして登録された **HttpAuthenticationFactory** を用いて **java.security.Provider** を実装する方法です。
- 2つ目は、**java.util.ServiceLoader** を使用して代わりにファクトリーを検出する方法です。これを行うには、**org.wildfly.security.http.HttpServerAuthenticationMechanismFactory** という名前のファイルを **META-INF/services** 以下に追加する必要があります。このファイルの内容には、ファクトリー実装の完全修飾クラス名のみが必要になります。

メカニズムは使用する準備が整ったモジュールとしてアプリケーションサーバーにインストールできます。

```
module add --name=org.wildfly.security.examples.custom-http --resources=/path/to/custom-http-mechanism.jar --dependencies=org.wildfly.security.elytron,javax.api
```

カスタム HTTP メカニズムの使用

1. カスタムモジュールを追加します。

```
/subsystem=elytron/service-loader-http-server-mechanism-factory=custom-factory:add(module=org.wildfly.security.examples.custom-http)
```

2. **http-authentication-factory** を追加して、メカニズムファクトリーを認証に使用される **security-domain** と結び付けます。

```
/subsystem=elytron/http-authentication-factory=custom-mechanism:add(http-server-
mechanism-factory=custom-factory,security-domain=ApplicationDomain,mechanism-
configurations=[{mechanism-name=custom-mechanism}])
```

3. **application-security-domain** リソースを更新し、新しい **http-authentication-factory** を使用するようになります。



注記

アプリケーションがデプロイされると、デフォルトで **other** セキュリティドメインを使用します。そのため、アプリケーションへのマッピングを追加して、Elytron HTTP 認証ファクトリーにマップする必要があります。

```
/subsystem=undertow/application-security-domain=other:add(http-
authentication-factory=application-http-authentication)
```

application-security-domain リソースを更新して、新しい **http-authentication-factory** を使用できるようになりました。

```
/subsystem=undertow/application-security-domain=other:write-attribute(name=http-
authentication-factory,value=custom-mechanism)
```

```
/subsystem=undertow/application-security-domain=other:write-attribute(name=override-
deployment-config,value=true)
```

上記のコマンドラインはデプロイメントの設定をオーバーライドすることに注意してください。そのため、デプロイメントが別のメカニズムを使用するように設定されていても **http-authentication-factory** からのメカニズムが使用されます。よって、デプロイメント自体の変更を必要としなくても、デプロイメント内で設定をオーバーライドしてカスタムメカニズムを利用することが可能です。

4. サーバーをリロードします。

```
reload
```

第11章 JAVA TRANSACTION API (JTA)

11.1. 概要

11.1.1. Java Transaction API (JTA) の概要

はじめに

ここでは、Java Transaction API (JTA) の基礎的な内容について取り上げます。

- [Java Transaction API \(JTA\)](#)
- [トランザクションライフサイクル](#)
- [JTA トランザクションの例](#)

11.2. トランザクションの概念

11.2.1. トランザクション

トランザクションは2つ以上のアクションで構成されており、アクションすべてが成功または失敗する必要があります。成功した場合はコミット、失敗した場合はロールバックが結果的に実行されます。ロールバックでは、トランザクションがコミットを試行する前に、各メンバーの状態が元の状態に戻ります。

よく設計されたトランザクションの通常の標準は Atomic, Consistent, Isolated, and Durable (ACID) です。

11.2.2. トランザクションの ACID プロパティ

ACID は 原子性 (**Atomicity**)、一貫性 (**Consistency**)、独立性 (**Isolation**)、永続性 (**Durability**) の略語です。通常、この用語はデータベースやトランザクション操作において使用されます。

原子性 (Atomicity)

トランザクションの原子性を保つには、すべてのトランザクションメンバーが同じ決定を行う必要があります。これらのメンバーはコミットまたはロールバックを行います。原子性が保たれない場合の結果はヒューリスティックな結果と呼ばれます。

一貫性

一貫性とは、データベーススキーマの観点から、データベースに書き込まれたデータが有効なデータであることを保証するという意味です。データベースあるいは他のデータソースは常に一貫した状態でなければなりません。一貫性のない状態の例には、操作が中断される前にデータの半分が書き込まれてしまったフィールドなどがあります。すべてのデータが書き込まれた場合や、書き込みが完了しなかった時に書き込みがロールバックされた場合に、一貫した状態となります。

分離

独立性とは、トランザクションのスコープ外のプロセスがデータを変更できないように、トランザクションで操作されたデータが変更前にロックされる必要があることを意味します。

持続性 (Durability)

持続性とは、トランザクションのメンバーがコミットするよう指示されてから外部で問題が発生した場合に、問題が解決されるとすべてのメンバーがトランザクションを継続してコミットできることを意味します。このような問題には、ハードウェア、ソフトウェア、ネットワーク、またはその他の関与するシステムが関連することがあります。

11.2.3. トランザクションコーディネーターまたはトランザクションマネージャー

JBoss EAP のトランザクションでは、トランザクションコーディネーターとトランザクションマネージャー (TM) という言葉は、ほとんど同じことを意味します。トランザクションコーディネーターという言葉は通常、分散 JTS トランザクションのコンテキストで使用されます。

JTA トランザクションでは、TM は JBoss EAP 内で実行され、2 フェーズコミットのプロトコルでトランザクションの参加者と通信します。

TM はトランザクションの参加者に対して、他のトランザクションの参加者の結果に従い、データをコミットするか、ロールバックするか指示します。こうすることで、確実にトランザクションが ACID 標準に準拠するようにします。

- [トランザクションの参加者](#)
- [トランザクションの ACID プロパティ](#)
- [2 フェーズコミットプロトコル](#)

11.2.4. トランザクションの参加者

トランザクションの参加者は、状態をコミットまたはロールバックできるトランザクション内のリソースであり、一般的にデータベースまたは JMS ブローカーを生成します。ただし、トランザクションインターフェースを実装することにより、アプリケーションコードがトランザクションの参加者として動作することもできます。トランザクションの各参加者は、状態をコミットまたはロールバックできるかどうかを独自に決定します。そして、すべての参加者がコミットできる場合のみ、トランザクション全体が成功します。コミットできない参加者がある場合は、各参加者がそれぞれの状態をロールバックし、トランザクション全体が失敗します。TM は、コミットおよびロールバック操作を調整し、トランザクションの結果を判断します。

11.2.5. Java Transaction API (JTA)

Java Transaction API (JTA) は Java Enterprise Edition 仕様の一部です。これは [JSR 907: Java™ Transaction API \(JTA\)](#) で定義されています。

JTA の実装は、JBoss EAP アプリケーションサーバーの Narayana プロジェクトに含まれる TM を使用して実行されます。TM により、単一のグローバルトランザクションを使用してアプリケーションがさまざまなリソース (データベースや JMS ブローカーなど) を割り当てできるようになります。グローバルトランザクションは XA トランザクションと呼ばれます。一般的に、このようなトランザクションには XA 機能を持つリソースが含まれますが、XA 以外のリソースがグローバルトランザクションに含まれることもあります。非 XA リソースを XA 対応リソースとして動作させるのに役に立つ複数の最適化があります。詳細は、[「1 フェーズコミット \(IPC\) の LRCO 最適化」](#) を参照してください。

本書では、JTA という用語は以下の 2 つのことを意味します。

1. Java EE 仕様によって定義される Java Transaction API。
2. TM がトランザクションをどのように処理するかを示します。

TM は JTA トランザクションモードで動作し、データはメモリーによって共有されます。また、トランザクションコンテキストはリモート EJB 呼び出しによって転送されます。JTS モードでは、データは CORBA (Common Object Request Broker Architecture) メッセージを送信して共有され、トランザクションコンテキストは IIOP 呼び出しによって転送されます。複数の JBoss EAP サーバー上におけるトランザクションの分散は両方のモードでサポートされます。

- [分散トランザクション](#)

- [XA データソースおよび XA トランザクション](#)

11.2.6. Java Transaction Service (JTS)

Java Transaction Service (JTS) は、Object Transaction Service (OTS) と Java のマッピングです。Java EE アプリケーションは JTA API を使用してトランザクションを管理します。JTA API はトランザクションマネージャーが JTS モードに切り替わったときに JTS トランザクション実装と対話します。JTS は IIOP プロトコル上で動作します。JTS を使用するトランザクションマネージャーは Object Request Broker (ORB) と呼ばれるプロセスと Common Object Request Broker Architecture (CORBA) と呼ばれる通信標準を使用してお互いに通信します。詳細は、JBoss EAP 『[設定ガイド](#)』の「[ORB 設定](#)」を参照してください。

アプリケーションの観点で JTA API を使用すると、JTS トランザクションは JTA トランザクションと同じように動作します。



注記

JBoss EAP に含まれる JTS の実装は、分散トランザクションをサポートします。完全準拠の JTS トランザクションとの違いは、外部のサードパーティー ORB との相互運用性です。この機能は、JBoss EAP ではサポートされません。サポートされる設定では、複数の JBoss EAP コンテナでのみトランザクションが分散されます。

11.2.7. XML トランザクションサービス

XML トランザクションサービス (XTS) コンポーネントは、ビジネストランザクションのプライベートおよびパブリック web サービスの調整をサポートします。XTS を使用すると、複雑なビジネストランザクションを制御され信頼できる状態で調整できます。XTS API は WS-Coordination、WS-Atomic Transaction、および WS-Business Activity プロトコルを基にしたトランザクションコーディネーションモデルをサポートします。

11.2.7.1. XTS によって使用されるプロトコルの概要

WS-Coordination (WS-C) 仕様は、異なるコーディネーションプロトコルがプラグインできるようにするフレームワークを定義し、クライアント、サービス、および参加者の間で作業を調整します。

WS-Transaction (WS-T) プロトコルは、WS-C によって提供されるコーディネーションフレームワークを利用する WS-Atomic Transaction (WS-AT) および WS-Business Activity (WS-BA) の 2 つのトランザクションコーディネーションプロトコルで構成されます。WS-T は、既存の従来のトランザクション処理システムを統一するために開発され、これらのシステム間で確実に通信が行われるようにします。

11.2.7.2. Web Services-Atomic Transaction (WS-AT) プロセス

アトミックトランザクション (AT) は、ACID セマンティックが適切である場合に短期間の対話をサポートするよう設計されています。AT の範囲内では、web サービスは通常 WS-T の制御下でブリッジングを用いてデータベースやメッセージキューなどの XA リソースにアクセスします。トランザクションが終了すると、参加者は AT の決定結果を XA リソースに伝搬し、各参加者によって適切なコミットまたはロールバックが実行されます。

11.2.7.2.1. アトミックトランザクション (AT) プロセス

1. AT を開始する際、クライアントは最初に WS-T をサポートする WS-C Activation Coordinator web サービスを見つけます。

2. クライアントは、<http://schemas.xmlsoap.org/ws/2004/10/wsat> をコーディネーション型として指定して、WS-C `CreateCoordinationContext` メッセージをサービスに送信します。
3. クライアントは適切な WS-T コンテキストをアクティベーションサービスから受け取ります。
4. `CreateCoordinationContext` メッセージの応答であるトランザクションコンテキストの `CoordinationType` 要素は、WS-AT ネームスペース <http://schemas.xmlsoap.org/ws/2004/10/wsat> に設定されています。また、参加者を登録できるアトミックトランザクションコーディネーターエンドポイントである WS-C Registration Service への参照も含まれます。
5. クライアントは通常、続いて Web サービスの呼び出しを行い、Web サービスによるすべての変更をコミットまたはロールバックしてトランザクションを完了します。完了できるようにするには、エンドポイントがコーディネーションコンテキストで返された登録サービスに登録メッセージを送信し、クライアントを完了プロトコルの参加者として登録する必要があります。
6. クライアントを登録したら、クライアントアプリケーションは web サービスと対話してビジネスレベルの作業を達成します。クライアントは、ビジネス web サービスが呼び出されるごとに、トランザクションコンテキストを SOAP ヘッダーブロックの挿入し、各呼び出しがトランザクションによって暗黙的にスコープ付けされます。WS-AT 対応 web サービスをサポートするツールキットは、SOAP ヘッダーブロックで見つかったコンテキストをバックエンド操作と関連付ける機能を提供します。これにより、web サービスによる変更がクライアントと同じトランザクションの範囲内で行われるようにし、トランザクションコーディネーターによるコミットまたはロールバックの対象になるようにします。
7. 必要なアプリケーションの作業がすべて完了したら、クライアントはサービス状態の変更を永続する目的でトランザクションを終了することができます。完了参加者は、トランザクションをコミットまたはロールバックするようコーディネーターに指示します。コミットまたはロールバック操作が完了すると、トランザクションの結果を示すために状態が参加者に返されます。

詳細は、『Naryana Project Documentation』の「[WS-Coordination](#)」を参照してください。

11.2.7.2.2. Microsoft .NET クライアントとの WS-AT の相互運用性

WS-AT 仕様の .NET 実装の違いにより、`xts` サブシステムと Microsoft .NET クライアントとの通信に問題が発生することがあります。WS-AT 仕様の .NET 実装はすべての呼び出しが非同期になるよう強制します。

.NET クライアントとの相互運用性を有効にするため、JBoss EAP の `xts` サブシステムでは非同期登録のオプションを利用できます。XTS 非同期登録はデフォルトでは無効になっており、必要な場合のみ有効にする必要があります。

非同期登録を有効にして .NET クライアントとの WS-AT の相互運用性を維持するには、以下の管理 CLI コマンドを実行します。

```
/subsystem=xts:write-attribute(name=async-registration, value=true)
```

11.2.7.3. Web Services-Business Activity (WS-BA) プロセス

Web Services-Business Activity (WS-BA) は、既存のビジネスプロセスおよびワークフローシステムがプロプライエタリーメカニズムをラップし、実装およびビジネス境界全体で相互運用できるようにする、web サービスアプリケーションのプロトコルを定義します。

要求時のみ参加者が状態をトランザクションコーディネーターに伝える WS-AT プロトコルモデルとは

異なり、WS-BA 内の子アクティビティーは要求を待たずに結果を直接コーディネーターに指定できます。参加者はいつでもアクティビティーを終了するかコーディネーターへ失敗を通知するかを選択できます。失敗を特定するためにトランザクションの最後まで待たずに、通知を使用してゴールを編集し、処理を継続できるため、この機能はタスクが失敗したときに便利です。

11.2.7.3.1. WS-BA プロセス

1. サービスは作業をするよう要求されます。
2. これらのサービスに作業を元に戻す機能があるのであれば、WS-BA が後でその作業の取り消しを決定した場合に備えて WS-BA に通知します。WS-BA に障害が発生した場合は、元に戻す **undo** 動作を実行するようサービスに指示することができます。

WS-BA プロトコルは補正ベースのトランザクションモデルを利用します。ビジネスアクティビティーの参加が作業を完了すると、アクティビティーを終了することを選択できます。この選択は、その後のロールバックを許可しません。この代わりに、参加者はアクティビティーを完了し、後で別の参加者が障害をコーディネーターに通知した場合に、行った作業を補正できることをコーディネーターに伝えることができます。この場合、コーディネーターは終了していない各参加者に障害を補正するよう要求し、適切であると見なされる補正アクションを実行する機会を与えます。すべての参加者が障害なしで終了または完了した場合、コーディネーターは完了した各参加者に対してアクティビティーがクローズしたことを通知します。

詳細は、『Naryana Project Documentation』の「[WS-Coordination](#)」を参照してください。

11.2.7.4. トランザクションブリッジの概要

トランザクションブリッジは、Java EE と WS-T ドメインをリンクするプロセスを説明します。トランザクションブリッジコンポーネントである **txbridge** は双方向のリンクを提供し、トランザクションのいずれの型も、別の型と使用するよう設計されているビジネスロジックを含めることができます。ブリッジによって使用される技術は、介入とプロトコルマッピングの組み合わせです。

トランザクションブリッジでは、介入されるコーディネーターは既存のトランザクションに登録され、プロトコルマッピングの追加タスクを実行します。これらのトランザクション型が異なっても、その親コーディネーターに対してはネイティブトランザクション型のリソースとして見られ、その子に対してはネイティブトランザクション型のコーディネーターとして見られます。

トランザクションブリッジは **org.jboss.jbossts.txbridge** パッケージとそのサブパッケージにあります。これは、クラスの2つのセットによって構成され、セットごとに各方向のブリッジ用になります。

詳細は、『Naryana Project Documentation』の「[TXBridge Guide](#)」を参照してください。

11.2.8. XA リソースおよび XA トランザクション

XA は eXtended Architecture を表し、複数のバックエンドデータストアを使用するトランザクションを定義するために X/Open Group によって開発されました。XA 標準は、グローバル TM とローカルリソースマネージャーとの間のインターフェースを定義します。XA では、4つの ACID プロパティーすべてを保持しながらアプリケーションサーバー、データベース、キャッシュ、メッセージキューなどの複数のリソースが同じトランザクションに参加できるようにします。4つの ACID プロパティーの1つは原子性であり、これは参加者の1つが変更のコミットに失敗した場合に他の参加者がトランザクションを中止し、トランザクションが発生する前の状態に戻すことを意味します。XA リソースは XA グローバルトランザクションに参加できるリソースです。

XA トランザクションは、複数のリソースにまたがることのできるトランザクションです。これには、コーディネートをを行う TM が関係します。この TM は、すべてが1つのグローバル XA トランザクションに関与する1つ以上のデータベースまたは他のトランザクションリソースを持ちます。

11.2.9. XA リカバリー

TM は X/Open XA 仕様を実装し、複数の XA リソースで XA トランザクションをサポートします。

XA リカバリーは、トランザクションの参加者であるリソースのいずれかがクラッシュしたり使用できなくなったりしても、トランザクションの影響を受けたすべてのリソースが確実に更新またはロールバックされるようにするプロセスのことです。JBoss EAP の範囲内では、XA データソース、JMS メッセージキュー、JCA リソースアダプターなどの XA リソースまたはサブシステムに対して、**transactions** サブシステムが XA リカバリーのメカニズムを提供します。

XA リカバリーはユーザーの介入がなくても実行されます。XA リカバリーに失敗すると、エラーがログ出力に記録されます。サポートが必要な場合は、Red Hat グローバルサポートサービスまでご連絡ください。XA リカバリープロセスは、デフォルトで 2 分ごとに開始される定期リカバリースレッドにより開始されます。定期リカバリースレッドにより、未完了のすべてのトランザクションが処理されます。



注記

未確定なトランザクションのリカバリーを完了するには 4-8 分ほどかかることがあります。これはリカバリープロセスを複数回実行する必要がある場合があるためです。

11.2.10. XA リカバリープロセスの制限

XA リカバリーには以下の制限があります。

- トランザクションログが正常にコミットされたトランザクションから消去されないことがあります。

XAResource のコミットメソッドが正常に完了し、トランザクションをコミットした後、コーディネーターがログをアップデートできるようになる前に JBoss EAP サーバーがクラッシュした場合、サーバーの再起動時に以下の警告メッセージが表示されることがあります。

```
ARJUNA016037: Could not find new XAResource to use for recovering non-serializable
XAResource XAResourceRecord
```

これは、リカバリー時に JBoss トランザクションマネージャー (TM) はログのトランザクション参加者を確認し、コミットを再試行しようとするからです。最終的に、JBoss TM はリソースがコミットされたと思われ、コミットを再試行しなくなります。このような場合、トランザクションはコミットされデータの損失はないため、警告を無視しても問題ありません。

警告が表示されないようにするには、**com.arjuna.ats.jta.xaAssumeRecoveryComplete** プロパティの値を **true** に設定します。このプロパティは、登録された **XAResourceRecovery** インスタンスから新しい **XAResource** インスタンスが見つからないとチェックされます。**true** に設定すると、リカバリーで、以前のコミットの試行が成功したと思われ、これ以上リカバリーを試行しなくてもインスタンスをログから削除できます。このプロパティはグローバルであり、適切に使用しないと **XAResource** インスタンスがコミットされていない状態のままになるため、注意して使用する必要があります。



注記

JBoss EAP 7.2 には、トランザクションが正常にコミットされた後にトランザクションログを消去する拡張機能が実装されているため、上記の状況は頻繁に発生しません。

- XAResource.prepare()** の最後にサーバーがクラッシュすると、JTS トランザクションに対するロールバックは呼び出されません。

XAResource.prepare() メソッド呼び出しの完了後に JBoss EAP サーバーがクラッシュすると、参加している **XAResource** インスタンスはすべて準備済みの状態でロックされ、サーバーの再起動時にその状態を維持します。トランザクションがタイムアウトするか、データベース管理者が手動でリソースをロールバックしてトランザクションログを消去するまで、トランザクションはロールバックされずリソースはロックされたままになります。詳細は、[「https://issues.jboss.org/browse/JBTM-2124」](https://issues.jboss.org/browse/JBTM-2124) を参照してください。

- 周期リカバリーはコミットされたトランザクションで発生する可能性があります。サーバーに過剰な負荷がかかっている場合、サーバーログには以下の警告メッセージとそれに続くスタックトレースが含まれる場合があります。

```
ARJUNA016027: Local XARecoveryModule.xaRecovery got XA exception
XAException.XAER_NOTA: javax.transaction.xa.XAException
```

負荷が大きい場合、トランザクションにかかる処理時間が周期的リカバリープロセスのアクティビティーと重なることがあります。周期的リカバリープロセスは進行中のトランザクションを検出し、ロールバックを開始しようとしませんが、トランザクションは完了するまで続行されます。周期的リカバリーはロールバックの開始に失敗し、ロールバックの失敗がサーバーログに記録されます。この問題の根本的な原因は、今後のリリースで修正される予定ですが、現時点では回避方法を使用できます。

com.arjuna.ats.jta.orphanSafetyInterval プロパティの値をデフォルト値の **10000** ミリ秒よりも大きくし、リカバリープロセスの2つのフェーズの間隔を増やします。**40000** ミリ秒の値が推奨されます。この設定では問題は解決されないことに注意してください。問題が発生して警告メッセージがログに記録される可能性が減少します。詳細は、[「https://developer.jboss.org/thread/266729」](https://developer.jboss.org/thread/266729) を参照してください。

11.2.11. 2 フェーズコミットプロトコル

2 フェーズコミット (2PC) プロトコルは、トランザクションの結果を決定するアルゴリズムを参照します。2PC は XA トランザクションを完了するプロセスとしてトランザクションマネージャー (TM) によって開始されます。

フェーズ 1: 準備

最初のフェーズでは、トランザクションをコミットできるか、あるいはロールバックする必要があるかをトランザクションの参加者がトランザクションコーディネーターに通知します。

フェーズ 2: コミット

2 番目のフェーズでは、トランザクションコーディネーターがトランザクション全体をコミットするか、またはロールバックするかを決定します。いずれの参加者もコミットできない場合、トランザクションはロールバックしなければなりません。それ以外の場合、トランザクションはコミットできます。コーディネーターは何を行うかをリソースに指示し、リソースはその完了時にコーディネーターに通知します。この時点で、トランザクションは完了します。

11.2.12. トランザクションタイムアウト

原子性を確保し、トランザクションを ACID 標準に準拠させるために、トランザクションの一部が長期間実行される場合があります。トランザクションの参加者は、コミット時にデータベーステーブルまたはキュー内のメッセージの一部である XA リソースをロックする必要があります。また、TM は各トランザクション参加者からの応答を待ってからすべての参加者にコミットあるいはロールバックの指示を出す必要があります。ハードウェアあるいはネットワークの障害のため、リソースが永久にロックされることがあります。

トランザクションのタイムアウトをトランザクションと関連付け、ライフサイクルを制御することができます。タイムアウトのしきい値がトランザクションのコミットあるいはロールバック前に渡された場合、タイムアウトにより、自動的にトランザクションがロールバックされます。

トランザクションサブシステム全体に対しデフォルトのタイムアウト値を設定できます。または、デフォルトのタイムアウト値を無効にし、トランザクションごとにタイムアウトを指定できます。

11.2.13. 分散トランザクション

分散トランザクションは、複数の JBoss EAP サーバー上に参加者が存在するトランザクションです。Java Transaction Service (JTS) 仕様では、異なるベンダーのアプリケーションサーバー間で JTS トランザクションを分散可能にすることが規定されています。Java Transaction API (JTA) はこれを定義していませんが、JBoss EAP は JBoss EAP サーバー間での分散 JTA トランザクションをサポートしています。



注記

異なるベンダーのサーバー間でのトランザクション分散はサポートされません。



注記

他のベンダーのアプリケーションサーバーのドキュメントでは、分散トランザクションという用語が XA トランザクションを意味することがあります。JBoss EAP のドキュメントでは、複数の JBoss EAP アプリケーションサーバー間で分散されるトランザクションを分散トランザクションと呼びます。また、本書では、異なるリソースで構成されるトランザクション (データベースリソースや JMS リソースなど) を XA トランザクションと呼びます。詳細は、「[Java Transaction Service \(JTS\)](#)」および「[XA データソースおよび XA トランザクション](#)」を参照してください。

11.2.14. ORB 移植性 API

Object Request Broker (ORB) とは、複数のアプリケーションサーバーで分散されるトランザクションの参加者、コーディネーター、リソース、および他のサービスにメッセージを送受信するプロセスのことです。ORB は標準的なインターフェース記述言語 (IDL) を使用してメッセージを通信し解釈します。Common Object Request Broker Architecture (CORBA) は JBoss EAP の ORB によって使用される IDL です。

ORB を使用する主なタイプのサービスは、Java Transaction Service (JTS) 仕様を使用する分散 Java トランザクションのシステムです。特に、レガシーシステムなどのその他のシステムでは、通信にリモートエンタープライズ JavaBeans、JAX-WS、または JAX-RS Web サービスなどのその他のメカニズムではなく ORB を使用できます。

ORB 移植性 API は ORB とやりとりするメカニズムを提供します。この API は ORB への参照を取得するメソッドや、ORB からの受信接続をリッスンするモードにアプリケーションを置くメソッドを提供します。API のメソッドの一部はすべての ORB によってサポートされていません。このような場合、例外が発生します。

API は 2 つの異なるクラスによって構成されます。

- **com.arjuna.orbportability.orb**
- **com.arjuna.orbportability.oa**

ORB Portability API に含まれるメソッドとプロパティの詳細は、[Red Hat カスタマーポータル](#)の JBoss EAP Javadocs バンドルを参照してください。

11.3. トランザクションの最適化

11.3.1. トランザクション最適化の概要

JBoss EAP のトランザクションマネージャー (TM) には、アプリケーションで利用できる複数の最適化機能が含まれています。

最適化機能は、特定のケースで 2 フェーズコミットプロトコルを拡張するために使用します。一般的に、TM によって、2 フェーズコミットを介して渡されるグローバルトランザクションが開始されます。ただし、これらのトランザクションを最適化する場合、特定のケースでは TM によって完全な 2 フェーズコミットを行う必要がないため、処理は速くなります。

TM により使用される別の最適化機能は、以下で詳細に説明されています。

- [1 フェーズコミット \(1PC\) の LRCO 最適化](#)
- [推定中止 \(presumed-abort\) の最適化](#)
- [読み取り専用の最適化](#)

11.3.2. 1 フェーズコミット (1PC) の LRCO 最適化

1 フェーズコミット (1PC)

トランザクションでは、2 フェーズコミットプロトコル (2PC) がより一般的に使用されますが、両フェーズに対応する必要がなかったり、対応できない場合もあります。そのような場合、1 フェーズコミット (1PC) プロトコルを使用できます。1 フェーズコミットプロトコルは、XA または非 XA リソースの 1 つがグローバルトランザクションの一部である場合に使用されます。

一般的に、準備フェーズでは、第 2 フェーズが処理されるまでリソースがロックされます。1 フェーズコミットは、準備フェーズが省略され、コミットのみがリソースに対して処理されることを意味します。指定されない場合は、グローバルトランザクションに参加者が 1 つだけ含まれるときに 1 フェーズコミット最適化機能が自動的に使用されます。

最終リソースコミット最適化 (LRCO: Last Resource Commit Optimization)

非 XA データソースが XA トランザクションに参加する場合は、最終リソースコミット最適化 (LRCO) と呼ばれる最適化機能を使用されます。このプロトコルにより、ほとんどのトランザクションは正常に完了しますが、一部のエラーによってトランザクションの結果の一貫性が失われることがあります。そのため、この方法は最終手段として使用してください。

非 XA リソースは準備フェーズの終了時に処理され、コミットが試行されます。コミットに成功すると、トランザクションログが書き込まれ、残りのリソースがコミットフェーズに移動します。最終リソースがコミットに失敗すると、トランザクションはロールバックされます。

ローカルの TX データソースが 1 つのみトランザクションで使用されると、LRCO が自動的に適用されます。

これまでは、LRCO メソッドを使用して非 XA リソースを XA トランザクションに追加していました。ただし、この場合は LRCO が失敗する可能性があります。LRCO メソッドを用いて非 XA リソースを XA トランザクションに追加する手順は次のとおりです。

1. XA トランザクションを準備します。
2. LRCO をコミットします。
3. トランザクションログを書き込みます。

4. XA トランザクションをコミットします。

手順2と手順3の間でクラッシュした場合、データが不整合になり、XA トランザクションをコミットできなくなることがあります。データの不整合が発生する理由は、LRCO 非 XA リソースがコミットされても、XA リソースの準備に関する情報が記録されなかったことです。リカバリーマネージャーはサーバーの起動後にリソースをロールバックします。CMR (Commit Markable Resource) ではこの制限がなくなり、非 XA リソースが確実に XA トランザクションに登録できるようになります。



注記

CMR は、特別な LRCO 最適化機能であり、データソースのみに使用する必要があります。非 XA リソースには適していません。

- [2 フェーズコミットプロトコル](#)

11.3.2.1. Commit Markable Resource (CMR)

概要

Commit Markable Resource (CMR) インターフェースを使用してリソースマネージャーへのアクセスを設定すると、非 XA データソースを XA (2PC) トランザクションに安定的に登録できます。これは、非 XA リソースを完全にリカバリー可能にする LRCO アルゴリズムの実装です。

CMR を設定するには、以下のことを行う必要があります。

1. データベースでテーブルを作成します。
2. データソースを接続可能にします。
3. **transactions** サブシステムへの参照を追加します。

データベースでのテーブルの作成

トランザクションには CMR リソースを1つだけ含めることができます。以下の例と似た SQL を使用してテーブルを作成できます。

```
SELECT xid,actionuid FROM _tableName_ WHERE transactionManagerID IN (String[])
DELETE FROM _tableName_ WHERE xid IN (byte[])
INSERT INTO _tableName_ (xid, transactionManagerID, actionuid) VALUES (byte[],String,byte[])
```

以下は、さまざまなデータベース管理システムのテーブルを作成するための SQL 構文の例になります。

例: Sybase のテーブル作成構文

```
CREATE TABLE xids (xid varbinary(144), transactionManagerID varchar(64), actionuid
varbinary(28))
```

例: Oracle のテーブル作成構文

```
CREATE TABLE xids (xid RAW(144), transactionManagerID varchar(64), actionuid RAW(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

例: IBM のテーブル作成構文

```
CREATE TABLE xids (xid VARCHAR(255) for bit data not null, transactionManagerID
varchar(64), actionuid VARCHAR(255) for bit data not null)
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

例: SQL Server のテーブル作成構文

```
CREATE TABLE xids (xid varbinary(144), transactionManagerID varchar(64), actionuid
varbinary(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

例: PostgreSQL のテーブル作成構文

```
CREATE TABLE xids (xid bytea, transactionManagerID varchar(64), actionuid bytea)
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

例: MariaDB のテーブル作成構文

```
CREATE TABLE xids (xid BINARY(144), transactionManagerID varchar(64), actionuid BINARY(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

例: MySQL のテーブル作成構文

```
CREATE TABLE xids (xid VARCHAR(255), transactionManagerID varchar(64), actionuid
VARCHAR(255))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

データソースを接続可能にする

デフォルトでは、CMR 機能はデータソースに対して無効になっています。有効にするには、データソースの設定を作成または変更し、**connectable** 属性を **true** に設定する必要があります。以下に、サーバーの XML 設定ファイルのデータソースセクションの例を示します。

```
<datasource enabled="true" jndi-name="java:jboss/datasources/ConnectableDS" pool-
name="ConnectableDS" jta="true" use-java-context="true" connectable="true"/>
```



注記

この機能は XA データソースには適用されません。

また、以下のように管理 CLI を使用してリソースマネージャーを CMR として有効にすることもできます。

```
/subsystem=datasources/data-source=ConnectableDS:add(enabled="true", jndi-
name="java:jboss/datasources/ConnectableDS", jta="true", use-java-context="true",
connectable="true", connection-url="validConnectionURL", exception-sorter-class-
name="org.jboss.jca.adapters.jdbc.extensions.mssql.MSSQLExceptionSorter", driver-name="mssql")
```

このコマンドは、サーバー設定ファイルの **datasources** セクションで以下の XML を生成します。

```
<datasource jta="true" jndi-name="java:jboss/datasources/ConnectableDS" pool-
name="ConnectableDS" enabled="true" use-java-context="true" connectable="true">
  <connection-url>validConnectionURL</connection-url>
```

```

<driver>mssql</driver>
<validation>
  <exception-sorter class-
name="org.jboss.jca.adapters.jdbc.extensions.mssql.MSSQLExceptionSorter"/>
</validation>
</datasource>

```



注記

データソースには、有効なドライバーが定義されている必要があります。上記の例では、**mssql** が **driver-name** として使用されますが、**mssql** は存在しません。詳細は、JBoss EAP 『[設定ガイド](#)』の「[MySQL データソースの例](#)」を参照してください。



注記

データソース設定で **exception-sorter-class-name** パラメーターを使用します。詳細については、JBoss EAP 『[設定ガイド](#)』の「[データソース設定の例](#)」を参照してください。

新しい CMR 機能を使用するために既存のリソースを更新

CMR 機能を使用するために既存のデータソースのみを更新する必要がある場合は、**connectable** 属性を変更します。

```
/subsystem=datasources/data-source=ConnectableDS:write-attribute(name=connectable,value=true)
```

transactions サブシステムに参照を追加

transactions サブシステムは、以下のような **transactions** サブシステム設定セクションへのエントリーを用いて CMR 対応のデータソースを特定します。

```

<subsystem xmlns="urn:jboss:domain:transactions:5.0">
  ...
  <commit-markable-resources>
    <commit-markable-resource jndi-name="java:jboss/datasources/ConnectableDS">
      <xid-location name="xids" batch-size="100" immediate-cleanup="false"/>
    </commit-markable-resource>
  ...
  </commit-markable-resources>
</subsystem>

```

以下のように管理 CLI を使用すると同じ結果を実現できます。

```
/subsystem=transactions/commit-markable-
resource=java:jboss/datasources/ConnectableDS/:add(batch-size=100,immediate-
cleanup=false,name=xids)
```



注記

transactions サブシステムで CMR 参照を追加したら、サーバーを再起動する必要があります。

11.3.3. 推定中止 (presumed-abort) の最適化

トランザクションをロールバックする場合、この情報をローカルで記録し、エンリストされたすべての参加者に通知します。この通知は形式的なもので、トランザクションの結果には影響しません。すべての参加者が通知されると、このトランザクションに関する情報を削除できます。

トランザクションのステータスに対する後続の要求が行われる場合、利用可能な情報はありません。このような場合、要求側はトランザクションが中断され、ロールバックされたと見なします。推定中止 (presumed-abort) の最適化は、トランザクションがコミットの実行を決定するまで参加者に関する情報を永続化する必要がないことを意味します。これは、トランザクションがコミットの実行を決定する前に発生した障害はトランザクションの中止であると推定されるためです。

11.3.4. 読み取り専用の最適化

参加者は、準備するよう要求されると、トランザクション中に変更したデータがないことをコーディネーターに伝えることができます。参加者が最終的にどうなってもトランザクションに影響を与えることはないため、このような参加者にトランザクションの結果について通知する必要はありません。この読み取り専用の参加者はコミットプロトコルの第2フェーズから省略可能です。

11.4. トランザクションの結果

11.4.1. トランザクションの結果

可能なトランザクションの結果は次の3つになります。

コミット

トランザクションの参加者すべてがコミットできる場合、トランザクションコーディネーターはコミットの実行を指示します。詳細は、「[トランザクションのコミット](#)」を参照してください。

ロールバック

トランザクションの参加者のいずれかがコミットできなかつたり、トランザクションコーディネーターが参加者にコミットを指示できない場合は、トランザクションがロールバックされます。詳細は、「[トランザクションのロールバック](#)」を参照してください。

ヒューリスティックな結果

トランザクションの参加者の一部がコミットし、他の参加者がロールバックした場合をヒューリスティックな結果と呼びます。ヒューリスティックな結果には、人間の介入が必要になります。詳細は、「[ヒューリスティックな結果](#)」を参照してください。

11.4.2. トランザクションのコミット

トランザクションの参加者がコミットすると、新規の状態が永続化されます。新規の状態はトランザクションで作業を行った参加者により作成されます。トランザクションのメンバーがデータベースに記録を書き込む時などが最も一般的な例になります。

コミット後、トランザクションの情報はトランザクションコーディネーターから削除され、新たに書き込まれた状態が永続状態となります。

11.4.3. トランザクションのロールバック

トランザクションの参加者は、トランザクションの開始前に、状態を反映するために状態をリストアし、ロールバックを行います。ロールバック後の状態はトランザクション開始前の状態と同じになります。

11.4.4. ヒューリスティックな結果

ヒューリスティックな結果 (アトミックでない結果) は、トランザクションでの参加者の決定がトランザクションマネージャーのものとは異なる状況です。ヒューリスティックな結果が起こると、システムの整合性が保たれなくなることがあり、通常、解決に人的介入が必要になります。ヒューリスティックな結果に依存するようなコードは記述しないようにしてください。

通常、ヒューリスティックな結果は、2フェーズコミット (2PC) プロトコルの2番目のフェーズで発生します。まれに、この結果が1PCで発生することがあります。多くの場合、これは基盤のハードウェアまたは基盤のサーバーの通信サブシステムの障害によって引き起こされます。

ヒューリスティックな結果は、トランザクションマネージャーおよび完全なクラッシュリカバリーをしようとした場合でも、さまざまなサブシステムまたはリソースのタイムアウトによって可能になります。何らかの形の分散合意が必要なシステムでは、グローバルな結果という点でシステムのいくつかの部分に分岐する状況が発生することがあります。

ヒューリスティックな結果には4種類あります。

ヒューリスティックロールバック

コミット操作はリソースをコミットできませんでしたが、すべての参加者はロールバックでき、アトミックな結果が実現されました。

ヒューリスティックコミット

参加者のすべてが一方向的にコミットしたため、ロールバック操作に失敗します。たとえば、コーディネーターが正常にトランザクションを準備したにも関わらず、ログ更新の失敗などでコーディネーター側で障害が発生したため、ロールバックの実行を決定した場合などに発生します。暫定的に参加者がコミットの実行を決定する場合があります。

ヒューリスティック混合

一部の参加者がコミットし、その他の参加者はロールバックした状態です。

ヒューリスティックハザード

更新の一部の配置が不明な状態です。不明なものはすべてコミットまたはロールバックされています。

- [2フェーズコミットプロトコル](#)

11.4.5. JBoss Transactions エラーと例外

UserTransaction のメソッドによって発生する例外に関する詳細は、[UserTransaction](#) API Javadoc を参照してください。

11.5. トランザクションライフサイクルの概要

11.5.1. トランザクションライフサイクル

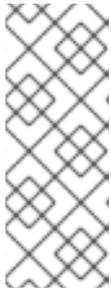
Java Transaction API (JTA) の詳細は、「[Java Transaction API \(JTA\)](#)」を参照してください。

リソースがトランザクションへの参加を要求すると、一連のイベントが開始されます。トランザクションマネージャー (TM) は、アプリケーションサーバー内に存在するプロセスであり、トランザクションを管理します。トランザクションの参加者は、トランザクションに参加するオブジェクトです。リソースは、データソース、JMS 接続ファクトリー、または他の JCA 接続です。

1. アプリケーションは新しいトランザクションを開始します。
トランザクションを開始するために、アプリケーションは JNDI から (EJB の場合はアノテーションから) **UserTransaction** クラスのインスタンスを取得します。**UserTransaction** インターフェイスには、トップレベルのトランザクションを開始、コミット、およびロールバックするメソッドが含まれています。新規作成されたトランザクションは、そのトランザクション

を呼び出すスレッドと自動的に関連付けされます。ネストされたトランザクションは JTA ではサポートされないため、すべてのトランザクションがトップレベルのトランザクションとなります。

UserTransaction.begin() メソッドが呼び出されると、EJB がトランザクションを開始します。このトランザクションのデフォルトの動作は **TransactionAttribute** アノテーションまたは **ejb.xml** 記述子の使用によって影響を受けることがあります。この時点以降に使用されたリソースは、このトランザクションと関連付けられます。2 つ以上のリソースが登録された場合、トランザクションは XA トランザクションになり、コミット時に 2 フェーズコミットプロトコルに参加します。



注記

デフォルトでは、トランザクションは EJB のアプリケーションコンテナによって駆動されます。これは **Container Managed Transaction (CMT)** と呼ばれます。トランザクションをユーザー駆動にするには、**Transaction Management** を **Bean Managed Transaction (BMT)** に変更する必要があります。BMT では、**UserTransaction** オブジェクトはユーザーがトランザクションを管理するために使用できます。

2. アプリケーションが状態を変更します。
次の手順では、アプリケーションが作業を実行し、登録されたリソースに対してのみ状態を変更します。
3. アプリケーションはコミットまたはロールバックの実行を決定します。
アプリケーションの状態の変更が完了すると、アプリケーションはコミットするか、またはロールバックするかを決定します。これは、適切なメソッド (**UserTransaction.commit()** または **UserTransaction.rollback()**) を呼び出します。CMT の場合、このプロセスは自動的に駆動されますが、BMT の場合は、**UserTransaction** のメソッドコミットまたはロールバックを明示的に呼び出す必要があります。
4. TM がレコードからトランザクションを削除します。
コミットあるいはロールバックが完了すると、TM はレコードをクリーンアップし、トランザクションログからトランザクションに関する情報を削除します。

障害リカバリー

リソース、トランザクションの参加者、またはアプリケーションサーバーがクラッシュするか、使用できなくなった場合は、障害が解決され、リソースが再度使用できるようになったときに **Transaction Manager** がリカバリーを実行します。このプロセスは自動的に実行されます。詳細は、「[XA リカバリー](#)」を参照してください。

11.6. トランザクションサブシステムの設定

transactions サブシステムでは、統計、タイムアウト値、トランザクションロギングなどのトランザクションマネージャーのオプションを設定できます。トランザクションを管理し、トランザクションの統計を表示することもできます。

詳細は、JBoss EAP 『[設定ガイド](#)』の「[トランザクションの設定](#)」を参照してください。

11.7. 実際のトランザクションの使用

11.7.1. トランザクション使用の概要

次の手順は、アプリケーションでトランザクションを使用する必要がある場合に役に立ちます。

- トランザクションの制御
 - トランザクションの開始
 - トランザクションのコミット
 - トランザクションのロールバック
- トランザクションにおけるヒューリスティックな結果の処理方法
- トランザクションエラーの処理
- トランザクションに関するリファレンス

11.7.2. トランザクションの制御

はじめに

この手順のリストでは、JTS API を使用するアプリケーションでトランザクションを制御するさまざまな方法を概説します。

- トランザクションの開始
- トランザクションのコミット
- トランザクションのロールバック

11.7.2.1. トランザクションの開始

この手順では、新しいトランザクションの開始方法を示します。実行するトランザクションマネージャー (TM) が JTA または JTS のいずれかで設定されていれば API は同じになります。

1. **UserTransaction** のインスタンスを取得します。

@TransactionManagement(TransactionManagementType.BEAN) アノテーションを用いると、JNDI、インジェクション、または EJB のコンテキスト EJB が Bean 管理のトランザクションを使用する場合) を使用してインスタンスを取得できます。

- JNDI を使用してインスタンスを取得します。

```
new InitialContext().lookup("java:comp/UserTransaction")
```

- インジェクションを使用してインスタンスを取得します。

```
@Resource UserTransaction userTransaction;
```

- EJB コンテキストを使用してインスタンスを取得します。

- ステートレス/ステートフル Bean の場合

```
@Resource SessionContext ctx;
ctx.getUserTransaction();
```

- メッセージ駆動型 Bean の場合

■

```
@Resource MessageDrivenContext ctx;
ctx.getUserTransaction();
```

2. データソースに接続したら **UserTransaction.begin()** を呼び出します。

```
try {
    System.out.println("\nCreating connection to database: "+url);
    stmt = conn.createStatement(); // non-tx statement
    try {
        System.out.println("Starting top-level transaction.");
        userTransaction.begin();
        stmtx = conn.createStatement(); // will be a tx-statement
        ...
    }
}
```

結果

トランザクションが開始します。トランザクションをコミットまたはロールバックするまで、データソースの使用はすべてトランザクションになります。

完全な例は、「[JTA トランザクションの例](#)」を参照してください。



注記

EJB (CMT または BMT のいずれかと使用) の利点の1つは、コンテナがトランザクション処理の内部をすべて管理することです。そのため、ユーザーは JBoss EAP コンテナ間の XA トランザクションまたはトランザクションディストリビューションの一部であるトランザクションを処理する必要がありません。

11.7.2.1.1. ネストされたトランザクション

ネストされたトランザクションを用いると、アプリケーションは既存のトランザクションに埋め込まれるトランザクションを作成できます。このモデルでは、再帰的に複数のサブトランザクションをトランザクションに埋め込むことができます。親トランザクションをコミットまたはロールバックせずにサブトランザクションをコミットまたはロールバックできます。しかし、コミット操作の結果は、先祖のトランザクションがすべてコミットしたかどうかによって決まります。

実装固有の情報は、[Narayana プロジェクトドキュメンテーション](#)を参照してください。

ネストされたトランザクションは、JTS 仕様と使用した場合のみ利用できます。ネストされたトランザクションは JBoss EAP アプリケーションサーバーではサポートされない機能です。また、多くのデータベースベンダーがネストされたトランザクションをサポートしないため、ネストされたトランザクションをアプリケーションに追加する前にデータベースベンダーにお問い合わせください。

11.7.2.2. トランザクションのコミット

この手順では、Java Transaction API (JTA) を使用してトランザクションをコミットする方法を説明します。

要件

トランザクションは、コミットする前に開始する必要があります。トランザクションの開始方法は、「[トランザクションの開始](#)」を参照してください。

1. **UserTransaction** で **commit()** メソッドを呼び出します。

UserTransaction で `commit()` メソッドを呼び出すと、TM はトランザクションのコミットを実行します。

```
@Inject
private UserTransaction userTransaction;

public void updateTable(String key, String value) {
    EntityManager entityManager = entityManagerFactory.createEntityManager();
    try {
        userTransaction.begin();
        <!-- Perform some data manipulation using entityManager -->
        ...
        // Commit the transaction
        userTransaction.commit();
    } catch (Exception ex) {
        <!-- Log message or notify Web page -->
        ...
        try {
            userTransaction.rollback();
        } catch (SystemException se) {
            throw new RuntimeException(se);
        }
        throw new RuntimeException(ex);
    } finally {
        entityManager.close();
    }
}
```

2. CMT (Container Managed Transaction) を使用する場合は、手動でコミットする必要はありません。
Bean がコンテナ管理トランザクションを使用するよう設定すると、コンテナはコードで設定したアノテーションに基づいてトランザクションライフサイクルを管理します。

```
@PersistenceContext
private EntityManager em;

@Transactional(TransactionAttributeType.REQUIRED)
public void updateTable(String key, String value)
    <!-- Perform some data manipulation using entityManager -->
    ...
}
```

結果

データソースがコミットされ、トランザクションが終了します。そうでない場合は、例外が発生します。



注記

完全な例は、「[JTA トランザクションの例](#)」を参照してください。

11.7.2.3. トランザクションのロールバック

この手順では、Java Transaction API (JTA) を使用してトランザクションをロールバックする方法を説明します。

要件

トランザクションは、ロールバックする前に開始する必要があります。トランザクションの開始方法は、「[トランザクションの開始](#)」を参照してください。

1. **UserTransaction** で **rollback()** メソッドを呼び出します。
UserTransaction の **rollback()** メソッドを呼び出す場合、TM はトランザクションをロールバックし、データを前の状態に戻そうとします。

```
@Inject
private UserTransaction userTransaction;

public void updateTable(String key, String value)
    EntityManager entityManager = entityManagerFactory.createEntityManager();
    try {
        userTransaction.begin();
        <!-- Perform some data manipulation using entityManager -->
        ...
        // Commit the transaction
        userTransaction.commit();
    } catch (Exception ex) {
        <!-- Log message or notify Web page -->
        ...
        try {
            userTransaction.rollback();
        } catch (SystemException se) {
            throw new RuntimeException(se);
        }
        throw new RuntimeException(e);
    } finally {
        entityManager.close();
    }
}
```

2. CMT (Container Managed Transaction) を使用する場合は、トランザクションを手動でロールバックする必要はありません。
Bean がコンテナ管理トランザクションを使用するように設定すると、コンテナはコードで設定したアノテーションに基づいてトランザクションライフサイクルを管理します。



注記

CMT のロールバックは `RuntimeException` が発生すると実行されます。 `setRollbackOnly` メソッドを明示的に呼び出してロールバックを発生させることもできます。または、アプリケーション例外の `@ApplicationException(rollback=true)` を使用してロールバックできます。

結果

トランザクションは TM によりロールバックされます。



注記

完全な例は、「[JTA トランザクションの例](#)」を参照してください。

11.7.3. トランザクションにおけるヒューリスティックな結果の処理方法

ヒューリスティックなトランザクションの結果はよく発生するものではなく、通常は例外的な原因が存在します。ヒューリスティックという言葉は「手動」を意味し、こうした結果は通常手動で処理する必要があります。トランザクションのヒューリスティックな結果は、「[ヒューリスティックな結果](#)」を参照してください。

この手順では、Java Transaction API (JTA) を使用してトランザクションのヒューリスティックな結果を処理する方法を説明します。

- トランザクションのヒューリスティックな結果の原因は、リソースマネージャーがコミットまたはロールバックの実行を約束したにも関わらず、約束を守らなかったことにあります。原因としては、サードパーティーコンポーネント、サードパーティーコンポーネントと JBoss EAP 間の統合レイヤー、または JBoss EAP 自体の問題が考えられます。
ヒューリスティックなエラーの最も一般的な2つの原因は、環境での一時的な障害と、リソースマネージャー対応時のコーディングエラーです。
- 通常、環境内で一時的な障害が発生した場合は、ヒューリスティックなエラーを発見する前に気づくはずですが、原因としては、ネットワークの停止、ハードウェア障害、データベース障害、電源異常などが考えられます。
ストレステストの実施中にテスト環境でヒューリスティックな結果が発生した場合は、テスト環境の脆弱性を意味します。



警告

JBoss EAP は、障害発生時にヒューリスティックな状態ではないトランザクションを自動的にリカバリーしますが、ヒューリスティックなトランザクションのリカバリーは実行しません。

- 環境に明白な障害が発生していない場合や、ヒューリスティックな結果を簡単に再現できる場合は、おそらくコーディングエラーが原因です。サードパーティーベンダーに連絡して解決策があるかどうかを確認する必要があります。
JBoss EAP のトランザクションマネージャー自体に問題があることを疑う場合は、サポートチケットを作成する必要があります。
- 管理 CLI を使用して手動によるトランザクションのリカバリーを試すことができます。詳細は、JBoss EAP 『[Managing Transactions on JBoss EAP](#)』の「[Recovering a Transaction Participant](#)」を参照してください。
- トランザクションの結果を手作業で解決する処理は、障害の正確な状況によって異なります。環境に合わせて以下の手順を実行します。
 - 関係しているリソースマネージャーを特定します。
 - トランザクションマネージャーとリソースマネージャーの状態を調べます。
 - 関係するコンポーネントの1つまたは複数で、ログの消去とデータの照合を手動で強制します。
- テスト環境である場合や、データの整合性を気にしない場合は、トランザクションログを削除して JBoss EAP を再起動すると、ヒューリスティックな結果はなくなります。デフォルトのトラ

トランザクションログの場所はスタンドアロンサーバーでは `EAP_HOME/standalone/data/tx-object-store/` ディレクトリー、管理対象ドメインでは `EAP_HOME/domain/servers/SERVER_NAME/data/tx-object-store/` ディレクトリーになります。管理対象ドメインの場合、`SERVER_NAME` は、サーバーグループに参加している個々のサーバー名を示します。



注記

トランザクションログの場所は、使用中のオブジェクトストアや、**object-store-relative-to** および **object-store-path** パラメーターに設定された値にも左右されます。標準のシャドーログや Apache ActiveMQ Artemis ログなどのファイルシステムログの場合、デフォルトディレクトリーの場所が使用されますが、JDBC オブジェクトストアを使用する場合は、トランザクションログはデータベースに保存されます。

11.7.4. JTA トランザクションのエラー処理

11.7.4.1. トランザクションエラーの処理

トランザクションエラーは、多くの場合、タイミングに依存するため、解決するのが困難です。以下に、一部の一般的なエラーと、これらのエラーのトラブルシューティングに関するヒントを示します。



注記

これらのガイドラインはヒューリスティックエラーに適用されません。ヒューリスティックエラーが発生した場合は、[トランザクションにおけるヒューリスティックな結果の処理方法](#) を参照し、Red Hat グローバルサポートサービスまでお問い合わせください。

トランザクションがタイムアウトになったが、ビジネスロジックスレッドが認識しませんでした。

多くの場合、このようなエラーは、Hibernate がレイジーロードのデータベース接続を取得できない場合に発生します。頻繁に発生する場合は、タイムアウト値を増加できます。JBoss EAP『[設定ガイド](#)』の「[トランザクションマネージャーの設定](#)」を参照してください。

引き続き問題が解決されない場合、外部環境を調整して実行をさらに速くしたり、さらなる効率化のためにコードを再構築できることがあります。タイムアウトの問題が解消されない場合は、Red Hat グローバルサポートサービスにお問い合わせください。

トランザクションがすでにスレッドで実行されているか、`NotSupportedException` 例外が発生する

`NotSupportedException` 例外は、通常、JTA トランザクションをネストしようとし、ネストがサポートされていないことを示します。トランザクションをネストしようとしなないときは、多くの場合、スレッドプールタスクで別のトランザクションが開始されますが、トランザクションを中断または終了せずにタスクが終了します。

通常、アプリケーションはこれを自動的に処理する `UserTransaction` を使用します。その場合は、フレームワークに問題があることがあります。

コードで `TransactionManager` メソッドまたは `Transaction` メソッドを直接使用する場合は、トランザクションをコミットまたはロールバックするときに次の動作に注意してください。コードで `TransactionManager` メソッドを使用してトランザクションを制御する場合は、トランザクションをコミットまたはロールバックすると、現在のスレッドからトランザクションの関連付けが解除されます。ただし、コードで `Transaction` メソッドを使用する場合は、トランザクションが実行されているスレッドと関連付けられていないことがあり、スレッドプールに返す前に手動でスレッドからトランザクションの関連付けを解除する必要があります。

2番目のローカルリソースを登録することはできません。

このエラーは、2番目の非XAリソースをトランザクションに登録しようとした場合に、発生します。1つのトランザクションで複数のリソースが必要な場合、それらのリソースはXAである必要があります。

11.8. トランザクションに関するリファレンス

11.8.1. JTA トランザクションの例

この例では、JTA トランザクションを開始、コミット、およびロールバックする方法を示します。使用している環境に合わせて接続およびデータソースパラメーターを調整し、データベースで2つのテストテーブルをセットアップする必要があります。

```
public class JDBCExample {
    public static void main (String[] args) {
        Context ctx = new InitialContext();
        // Change these two lines to suit your environment.
        DataSource ds = (DataSource)ctx.lookup("jdbc/ExampleDS");
        Connection conn = ds.getConnection("testuser", "testpwd");
        Statement stmt = null; // Non-transactional statement
        Statement stmtx = null; // Transactional statement
        Properties dbProperties = new Properties();

        // Get a UserTransaction
        UserTransaction txn = new InitialContext().lookup("java:comp/UserTransaction");

        try {
            stmt = conn.createStatement(); // non-tx statement

            // Check the database connection.
            try {
                stmt.executeUpdate("DROP TABLE test_table");
                stmt.executeUpdate("DROP TABLE test_table2");
            }
            catch (Exception e) {
                throw new RuntimeException(e);
                // assume not in database.
            }

            try {
                stmt.executeUpdate("CREATE TABLE test_table (a INTEGER,b INTEGER)");
                stmt.executeUpdate("CREATE TABLE test_table2 (a INTEGER,b INTEGER)");
            }
            catch (Exception e) {
                throw new RuntimeException(e);
            }

            try {
                System.out.println("Starting top-level transaction.");

                txn.begin();

                stmtx = conn.createStatement(); // will be a tx-statement
```

```
// First, we try to roll back changes

System.out.println("\nAdding entries to table 1.");

stmtx.executeUpdate("INSERT INTO test_table (a, b) VALUES (1,2)");

ResultSet res1 = null;

System.out.println("\nInspecting table 1.");

res1 = stmtx.executeQuery("SELECT * FROM test_table");

while (res1.next()) {
    System.out.println("Column 1: "+res1.getInt(1));
    System.out.println("Column 2: "+res1.getInt(2));
}
System.out.println("\nAdding entries to table 2.");

stmtx.executeUpdate("INSERT INTO test_table2 (a, b) VALUES (3,4)");
res1 = stmtx.executeQuery("SELECT * FROM test_table2");

System.out.println("\nInspecting table 2.");

while (res1.next()) {
    System.out.println("Column 1: "+res1.getInt(1));
    System.out.println("Column 2: "+res1.getInt(2));
}

System.out.print("\nNow attempting to rollback changes.");

txn.rollback();

// Next, we try to commit changes
txn.begin();
stmtx = conn.createStatement();
System.out.println("\nAdding entries to table 1.");
stmtx.executeUpdate("INSERT INTO test_table (a, b) VALUES (1,2)");
ResultSet res2 = null;

System.out.println("\nNow checking state of table 1.");

res2 = stmtx.executeQuery("SELECT * FROM test_table");

while (res2.next()) {
    System.out.println("Column 1: "+res2.getInt(1));
    System.out.println("Column 2: "+res2.getInt(2));
}

System.out.println("\nNow checking state of table 2.");

stmtx = conn.createStatement();

res2 = stmtx.executeQuery("SELECT * FROM test_table2");

while (res2.next()) {
    System.out.println("Column 1: "+res2.getInt(1));
```

```
        System.out.println("Column 2: "+res2.getInt(2));
    }

    txn.commit();
}
catch (Exception ex) {
    throw new RuntimeException(ex);
}
}
catch (Exception sysEx) {
    sysEx.printStackTrace();
    System.exit(0);
}
}
}
```

11.8.2. トランザクション API ドキュメンテーション

トランザクション JTA API ドキュメンテーションは以下の場所で Javadoc として利用できます。

- UserTransaction - <http://docs.oracle.com/javaee/7/api/javax/transaction/UserTransaction.html>

Red Hat Developer Studio を使用してアプリケーションを開発する場合は、API ドキュメンテーションは **Help** メニューに含まれています。

第12章 JAVA 永続 API (JPA)

12.1. JAVA 永続 API (JPA) について

Java Persistence API (JPA) は、Java オブジェクトまたはクラスとリレーショナルデータベース間でデータのアクセス、永続化、および管理を行うための Java 仕様です。この JPA 仕様では、透過オブジェクトまたはリレーショナルマッピングのパラダイムが考慮されます。オブジェクトまたはリレーショナル永続化メカニズムに必要な基本的な API とメタデータが標準化されます。



注記

JPA 自体は製品ではなく仕様にすぎません。それ自体では永続化やその他の処理を実行できません。JPA はインターフェースセットにすぎず、実装を必要とします。

12.2. 単純な JPA アプリケーションの作成

Red Hat CodeReady Studio で単純な JPA アプリケーションを作成する場合は、以下の手順を実行します。

1. Red Hat CodeReady Studio で JPA プロジェクトを作成します。
 - a. Red Hat CodeReady Studio で、**File** → **New** → **Project** の順にクリックします。リストで **JPA** を見つけ、展開し、**JPA Project** を選択します。以下のダイアログが表示されます。

図12.1 新規 JPA プロジェクトダイアログ

New JPA Project

JPA Project

Configure JPA project settings.

Project name:

Project location

Use default location

Location:

Target runtime

JPA version

Configuration

A general starting point for a JPA application.

EAR membership

Add project to an EAR

EAR project name:

Working sets

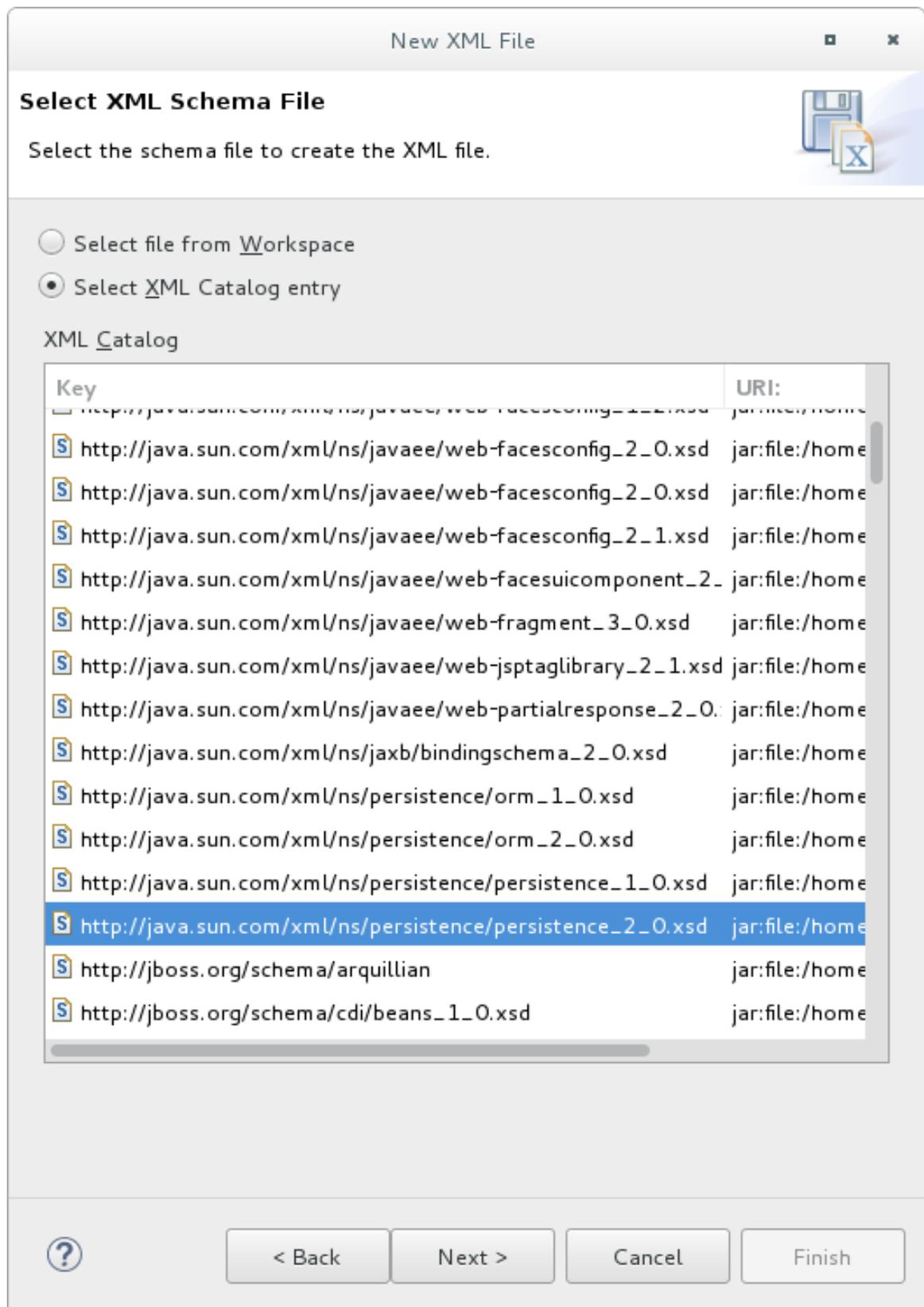
Add project to working sets

Working sets:

b. プロジェクト名を入力します。

- c. **Target runtime** を選択します。ターゲットランタイムがない場合は、『[Getting Started with Red Hat Developer Studio Tools](#)』の「**Downloading, Installing, and Setting Up JBoss EAP from within the IDE**」の手順に従って、新しいサーバーとランタイムを定義します。
 - d. **JPA version (JPA バージョン)** で 2.1 が選択されていることを確認します。
 - e. **Configuration (設定)** で **Basic JPA Configuration (基本的な JPA 設定)** を選択します。
 - f. **Finish** をクリックします。
 - g. 要求されたら、このタイプのプロジェクトを JPA パースペクティブウインドウに関連付けるかどうかを選択します。
2. 新しい永続性設定ファイルを作成および設定します。
 - a. Red Hat CodeReady Studio で EJB 3.x プロジェクトを開きます。
 - b. **Project Explorer (プロジェクトエクスプローラー)** パネルでプロジェクトルートディレクトリーを右クリックします。
 - c. **New (新規) → Other (その他)...** を選択します。
 - d. XML フォルダーから **XML File (XML ファイル)** を選択し、**Next (次へ)** をクリックします。
 - e. 親ディレクトリーとして **ejbModule/META-INF/** フォルダーを選択します。
 - f. ファイルの名前を **persistence.xml** と指定し、**Next (次へ)** をクリックします。
 - g. **Create XML file from an XML schema file (XML スキーマファイルから XML ファイルを作成)** を選択し、**Next (次へ)** をクリックします。
 - h. **Select XML Catalog entry (XML カタログエントリーを選択)** リストから **http://java.sun.com/xml/ns/persistence/persistence_2.0.xsd** を選択し、**Next (次へ)** をクリックします。

図12.2 永続 XML スキーマ



- i. **Finish (完了)** をクリックしてファイルを作成します。**persistence.xml** が **META-INF/** フォルダに作成され、設定可能な状態になります。

例: 永続設定ファイル

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_2.xsd"
version="2.2">
<persistence-unit name="example" transaction-type="JTA">
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
  <mapping-file>ormap.xml</mapping-file>
  <jar-file>TestApp.jar</jar-file>
  <class>org.test.Test</class>
  <shared-cache-mode>NONE</shared-cache-mode>
  <validation-mode>CALLBACK</validation-mode>
  <properties>
    <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
    <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
  </properties>
</persistence-unit>
</persistence>
```

12.3. JPA エンティティ

アプリケーションからデータベースへの接続を確立したら、データベースのデータを Java オブジェクトにマッピングできます。データベーステーブルに対してマップするために使用される Java オブジェクトはエンティティオブジェクトと呼ばれます。

エンティティは別のエンティティと関係を持ち、そのような関係はオブジェクトリレーショナルメタデータを介して表現されます。オブジェクトリレーショナルメタデータは、アノテーションを使用してエンティティクラスファイルに直接指定するか、アプリケーションに含まれる **persistence.xml** という XML 記述ファイルで指定できます。

Java オブジェクトからデータベースへのマッピングの概要は次のとおりです。

- Java クラスはデータベーステーブルにマップします。
- Java インスタンスはデータベース行にマップします。
- Java フィールドはデータベース列にマップします。

12.4. 永続コンテキスト

JPA 永続コンテキストには、永続プロバイダーによって管理されるエンティティが含まれます。永続コンテキストは、データソースと対話するための1次レベルトランザクションキャッシュのように動作します。これは、エンティティインスタンスおよびそれらのライフサイクルを管理します。ロードされたエンティティは、アプリケーションに返される前に永続コンテキストに置かれます。エンティティの変更も永続コンテキストに置かれ、トランザクションのコミットが実行されるとデータベースに保存されます。

コンテナ管理永続性 (CMP) コンテキストは、トランザクションにスコープ付けするか (トランザクションスコープの永続コンテキストと呼ばれます)、単一トランザクションを超えて拡張するライフタイムスコープを持つことができます (拡張永続コンテキストと呼ばれます)。enum データタイプを持つ **PersistenceContextType** プロパティは、コンテナ管理エンティティマネージャーの永続コンテキストライフタイムスコープを定義するために使用されます。永続コンテキストのライフタイムスコープは、**EntityManager** インスタンスの作成時に定義されます。

12.4.1. トランザクションスコープの永続コンテキスト

トランザクションスコープの永続コンテキストは、アクティブな JTA トランザクションと動作します。トランザクションがコミットすると、永続コンテキストはデータソースにフラッシュされます。エンティティオブジェクトはデタッチされますが、アプリケーションコードによって参照されることがあります。データソースに保存されることが想定されるエンティティの変更はすべてトランザクション中に行われる必要があります。トランザクション外部で読み取られるエンティティは、**EntityManager** 呼び出しが完了するとデタッチされます。

12.4.2. 拡張永続コンテキスト

拡張永続コンテキストは複数のトランザクションにまたがり、アクティブな JTA トランザクションがなくてもデータの変更をキューに置けるようにします。コンテナ管理の拡張永続コンテキストは、ステートフルセッション bean のみヘインジェクトできます。

12.5. JPA ENTITYMANAGER

JPA エンティティマネージャーは、永続コンテキストへの接続を表します。エンティティマネージャーを使用して永続コンテキストによって定義されるデータベースの読み書きが可能です。

永続コンテキストは、**javax.persistence** パッケージの Java アノテーション **@PersistenceContext** を介して提供されます。エンティティマネージャーは、Java クラス **javax.persistence.EntityManager** を介して提供されます。管理対象 bean では、以下のように **EntityManager** インスタンスをインジェクトできます。

例: エンティティマネージャーのインジェクション

```
@Stateless
public class UserBean {
    @PersistenceContext
    EntityManager entityManager;
    ...
}
```

12.5.1. アプリケーション管理の EntityManager

アプリケーション管理のエンティティマネージャーは基盤の永続プロバイダーである **org.hibernate.jpa.HibernatePersistenceProvider** への直接アクセスを提供します。アプリケーション管理のエンティティマネージャーの範囲は、アプリケーションによって作成された時からアプリケーションによってクローズされるまでです。**@PersistenceUnit** アノテーションを使用して、永続ユニットを **javax.persistence.EntityManagerFactory** インターフェースにインジェクトできます。これにより、アプリケーション管理のエンティティマネージャーが返されます。

アプリケーション管理のエンティティマネージャーは、特定の永続ユニットにて **EntityManager** インスタンスすべてで JTA トランザクションを使用して伝搬されていない永続コンテキストへアプリケーションがアクセスする必要があるときに使用できます。この場合、各 **EntityManager** インスタンスは新たに分離された永続コンテキストを作成します。**EntityManager** インスタンスと関連する **PersistenceContext** は、アプリケーションによって明示的に作成および破棄されます。**EntityManager** インスタンスはスレッドセーフではないため、**EntityManager** インスタンスを直接インジェクトできないときにアプリケーション管理のエンティティマネージャーを使用することもできます。**EntityManagerFactory** はスレッドセーフです。

例: アプリケーション管理のエンティティマネージャー

```
@PersistenceUnit
EntityManagerFactory emf;
```

```

EntityManager em;
@Resource
UserTransaction utx;
...
em = emf.createEntityManager();
try {
    utx.begin();
    em.persist(SomeEntity);
    em.merge(AnotherEntity);
    em.remove(ThirdEntity);
    utx.commit();
}
catch (Exception e) {
    utx.rollback();
}

```

12.5.2. コンテナ管理の EntityManager

コンテナ管理のエンティティマネージャーは、アプリケーションの基盤となる永続プロバイダーを管理します。トランザクションスコープの永続コンテキストまたは拡張永続コンテキストを使用できます。コンテナ管理のエンティティマネージャーは、必要に応じて基盤となる永続プロバイダーのインスタンスを作成します。基盤の永続プロバイダー `org.hibernate.jpa.HibernatePersistenceProvider` インスタンスが新たに作成されるたびに、新しい永続コンテキストも作成されます。

12.6. ENTITYMANAGER の利用

`/META-INF` ディレクトリに `persistence.xml` ファイルがある場合、エンティティマネージャーはロードされ、データベースにアクティブに接続されます。 **EntityManager** プロパティを使用して、エンティティマネージャーを JNDI にバインドし、エンティティを追加、更新、削除、およびクエリーできます。



重要

Hibernate でセキュリティーマネージャーの使用を計画している場合は、 **EntityManagerFactory** が JBoss EAP サーバーによってブートストラップされている場合のみ Hibernate がサポートすることに注意してください。 **EntityManagerFactory** または **SessionFactory** がアプリケーションによってブートストラップされている場合はサポートされません。セキュリティーマネージャーに関する詳細は、『[How to Configure Server Security](#)』の「[Java Security Manager](#)」を参照してください。

12.6.1. EntityManager の JNDI へのバインディング

デフォルトでは、JBoss EAP は **EntityManagerFactory** を JNDI にバインドしません。 `jboss.entity.manager.factory.jndi.name` プロパティを設定すると、アプリケーションの `persistence.xml` ファイルでこれを明示的に設定できます。このプロパティの値は、 **EntityManagerFactory** をバインドする JNDI の名前にする必要があります。

また、 `jboss.entity.manager.jndi.name` プロパティを使用すると、コンテナ管理でトランザクションスコープのエンティティマネージャーを JNDI にバインドすることもできます。

例: EntityManager および EntityManagerFactory の JNDI へのバインド

```

<property name="jboss.entity.manager.jndi.name" value="java:/MyEntityManager"/>
<property name="jboss.entity.manager.factory.jndi.name" value="java:/MyEntityManagerFactory"/>

```

例: EntityManager を使用したエンティティの格納

```
public User createUser(User user) {
    entityManager.persist(user);
    return user;
}
```

例: EntityManager を使用したエンティティの更新

```
public void updateUser(User user) {
    entityManager.merge(user);
}
```

例: EntityManager を使用したエンティティの削除

```
public void deleteUser(String user) {
    User user = findUser(username);
    if (user != null)
        entityManager.remove(user);
}
```

例: EntityManager を使用したエンティティのクエリー

```
public User findUser(String username) {
    CriteriaBuilder builder = entityManager.getCriteriaBuilder();
    CriteriaQuery<User> criteria = builder.createQuery(User.class);
    Root<User> root = criteria.from(User.class);
    TypedQuery<User> query = entityManager
        .createQuery(criteria.select(root).where(
            builder.equal(root.<String> get("username"), username)));
    try {
        return query.getSingleResult();
    }
    catch (NoResultException e) {
        return null;
    }
}
```

12.7. 永続ユニットのデプロイ

永続ユニットは、以下が含まれる論理グループです。

- エンティティマネージャーファクトリーおよびそのエンティティマネージャーの設定情報。
- エンティティマネージャーによって管理されるクラス。
- データベースへのクラスのマッピングを指定するメタデータのマッピング。

persistence.xml ファイルには、データソース名を含む永続ユニット設定が含まれています。**/META-INF/** ディレクトリーに **persistence.xml** ファイルが含まれる JAR ファイルまたはディレクトリーは、永続ユニットのルートと呼ばれます。

Java EE 環境では、永続ユニットのルートは以下の1つである必要があります。

- EJB-JAR ファイル
- WAR ファイルの **/WEB-INF/classes/** ディレクトリー
- WAR ファイルの **/WEB-INF/lib/** ディレクトリーにある JAR ファイル
- EAR ライブラリーディレクトリーの JAR ファイル
- アプリケーションクライアントの JAR ファイル

例: 永続設定ファイル

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_2.xsd"
  version="2.2">
  <persistence-unit name="example" transaction-type="JTA">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
    <mapping-file>ormap.xml</mapping-file>
    <jar-file>TestApp.jar</jar-file>
    <class>org.test.Test</class>
    <shared-cache-mode>NONE</shared-cache-mode>
    <validation-mode>CALLBACK</validation-mode>
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

12.8.2 次キャッシュ

12.8.1.2 次キャッシュ

2次キャッシュとは、アプリケーションセッション外部で永続化された情報を保持するローカルデータストアのことです。このキャッシュは永続プロバイダーにより管理され、アプリケーションとデータを分離することでランタイムを改善します。

JBoss EAP では、以下の目的のためにキャッシュがサポートされます。

- Web セッションのクラスタリング
- ステートフルセッション Bean のクラスタリング
- SSO クラスタリング
- Hibernate/JPA 2次キャッシュ



警告

各キャッシュコンテナは **repl** および **dist** キャッシュを定義します。これらのキャッシュは、ユーザーアプリケーションで直接使用しないでください。

12.8.1.1. デフォルトの 2 次キャッシュプロバイダー

Infinispan は、JBoss EAP のデフォルトの 2 次キャッシュプロバイダーです。Infinispan は、オプションのスキーマを持つ分散型のインメモリーキーと値のデータストアで、Apache License 2.0 で利用できます。

12.8.1.1.1. 永続ユニットでの 2 次レベルキャッシュの設定

永続ユニットの **shared-cache-mode** 要素を使用して 2 次レベルキャッシュを設定できます。

1. 「[単純な JPA アプリケーションの作成](#)」を参照して、Red Hat CodeReady Studio で **persistence.xml** ファイルを作成します。
2. 以下の内容を **persistence.xml** ファイルに追加します。

```
<persistence-unit name="...">
  (...) <!-- other configuration -->
  <shared-cache-mode>SHARED_CACHE_MODE</shared-cache-mode>
  <properties>
    <property name="hibernate.cache.use_second_level_cache" value="true" />
    <property name="hibernate.cache.use_query_cache" value="true" />
  </properties>
</persistence-unit>
```

SHARED_CACHE_MODE 要素には以下の値を指定できます。

- **ALL**: すべてのエンティティーがキャッシュ可能と見なされます。
- **NONE**: キャッシュ可能と見なされるエンティティーはありません。
- **ENABLE_SELECTIVE**: キャッシュ可能とマークされたエンティティーのみがキャッシュ可能と見なされます。
- **DISABLE_SELECTIVE**: 明示的にキャッシュ不可能であるとマークされたエンティティーを除くすべてのエンティティーがキャッシュ可能と見なされます。
- **UNSPECIFIED**: 動作は定義されません。プロバイダー固有のデフォルトは提供されます。

第13章 BEAN VALIDATION

13.1. BEAN VALIDATION

Bean Validation あるいは JavaBeans Validation は、Java オブジェクトのデータのバリデーションを行うモデルです。このモデルでは、組み込みのカスタムアノテーション制約を使い、アプリケーションデータの整合性を保ちます。また、メソッドおよびコンストラクターバリデーションも提供し、パラメーターおよび戻り値の制約を確保します。この仕様は「[JSR 380: Bean Validation 2.0](#)」で文書化されています。

Hibernate Validator は Bean Validation の JBoss EAP 実装です。また、JSR の参照実装です。

JBoss EAP は JSR 380 Bean Validation 2.0 仕様に完全準拠しています。また、Hibernate Validator によってこの仕様を追加機能が提供されます。

Bean Validation を初めて使用する場合は、JBoss EAP に同梱された **bean-validation** クイックスタートを参照してください。クイックスタートをダウンロードし、実行する方法は、JBoss EAP『[スタートガイド](#)』の「[クイックスタートサンプルの使用](#)」を参照してください。

JBoss EAP 7.2 には Hibernate Validator 6.0.x が含まれるようになりました。

Hibernate Validator 6.0.x の新機能

- Bean Validation 2.0 はエンティティおよびメソッドバリデーションのメタデータモデルおよび API を定義します。
メタデータのデフォルトのソースはアノテーションで、XML を使用してメタデータをオーバーライドおよび拡張することができます。

API は特定のアプリケーション階層やプログラミングモデルに関係しません。サーバー側のアプリケーションプログラミングとリッチクライアント Swing アプリケーションの開発の両方で利用できます。
- 本リリースの Hibernate Validator は、バグ修正が含まれるだけでなく、最も一般的なユースケースに対してパフォーマンスが向上されています。
- Bean Validation はバージョン 1.1 より、Bean Validation API を使用して任意 Java タイプのメソッドのパラメーターおよび戻り値に制約も適用できるようになりました。
- Hibernate Validator 6.0.x および Bean Validation 2.0 には Java 8 以上が必要になります。
詳細は「[Hibernate Validator 6.0.9.Final released](#)」および「[Hibernate Validator 6.0.13.Final - JSR 380 Reference Implementation: Reference Guide](#)」を参照してください。

13.2. バリデーション制約

13.2.1. バリデーション制約

バリデーション制約とは、フィールド、プロパティ、Bean などの Java 要素に適用するルールのことです。制約は通常、制限を設定する際に利用する一連の属性です。定義済みの制約がありますが、カスタムの制約も作成可能です。各制約は、アノテーション形式で表されます。

Hibernate Validator 用の同梱のバリデーション制約は、[Hibernate Validator の制約](#)にリストされています。

13.2.2. Hibernate Validator の制約



注記

該当する場合は、アプリケーションレベルの制約により、以下の表の **Hibernate Metadata Impact** 列で説明されているデータベースレベルの制約が作成されます。

Java 固有のバリデーション制約

以下の表には、**javax.validation.constraints** パッケージに含まれる Java 仕様で定義されたバリデーション制約が示されています。

アノテーション	プロパティタイプ	ランタイムチェック	Hibernate Metadata の影響
@AssertFalse	ブール値	メソッドが false に評価されていることを確認します。アノテーションではなくコードで表現される制約に便利です。	なし
@AssertTrue	ブール値	メソッドが true に評価されていることを確認します。アノテーションではなくコードで表現される制約に便利です。	なし
@Digits(integerDigits=1)	数値または数値の文字列表現	プロパティが integerDigits までの整数部と、 fractionalDigits までの小数部を持つ数字であるかを確認します。	カラムの精度とスケールを定義します。
@Future	日付またはカレンダー	未来の日付であるかを確認します。	なし
@Max(value=)	数値または数値の文字列表現	値が最大値以下であるかを確認します。	カラムに check 制約を追加します。
@Min(value=)	数値または数値の文字列表現	値が最小値以上であるかを確認します。	カラムに check 制約を追加します。
@NotNull		値が null でないかを確認します。	カラムが null でないかを確認します。
@Past	日付またはカレンダー	過去の日付であるかを確認します。	カラムに check 制約を追加します。

アノテーション	プロパティタイプ	ランタイムチェック	Hibernate Metadata の影響
@Pattern(regexp="regexp", flag=) or @Patterns({@Pattern(...)})	String	プロパティが一致フラグが指定された正規表現に一致するかどうかを確認します。 java.util.regex.Pattern を参照してください。	なし
@Size(min=, max=)	アレイ、コレクション、マップ	要素サイズが最小値以上で最大値以下であるかどうかを確認します。	なし
@Valid	オブジェクト	紐付けされたオブジェクトに再帰的にバリデーションを実行します。オブジェクトがコレクションかアレイの場合は、要素は再帰的に検証されます。また、オブジェクトがマップの場合、値要素が再帰的に検証されます。	なし



注記

パラメーター **@Valid** は、**javax.validation.constraints** パッケージに存在しますが Bean Validation 仕様の一部です。

Hibernate Validator 固有のバリデーション制約

以下の表には、**org.hibernate.validator.constraints** パッケージに含まれるベンダー固有のバリデーション制約が含まれます。

アノテーション	プロパティタイプ	ランタイムチェック	Hibernate Metadata の影響
@Length(min=, max=)	String	文字列の長さが指定の範囲と一致するかを確認します。	カラムの長さを最大に設定します。
@CreditCardNumber	String	文字列が正規の形式のクレジットカード番号であるかどうかを確認します (Luhn アルゴリズムの派生)。	なし

アノテーション	プロパティタイプ	ランタイムチェック	Hibernate Metadata の影響
@EAN	String	文字列が正しくフォーマットされた EAN あるいは UPC-A コードであるかを確認します。	なし
@Email	String	文字列がメールアドレスの仕様に準拠するかどうかを確認します。	なし
@NotEmpty		文字列が null あるいは空でないかを確認します。接続が null あるいは空でないかを確認します。	カラムは文字列の null ではありません。
@Range(min=, max=)	数値または数値の文字列表現	値が最小値以上で最大値以下であるかどうかを確認します。	カラムに check 制約を追加します。

13.2.3. カスタム制約を使用した Bean Validation

Bean Validation API は、**@NotNull** や **@Size** などの標準の制約アノテーションのセットを定義します。しかし、これらの事前定義された制約が十分でない場合、バリデーションの要件に合ったカスタム制約を簡単に作成できます。

Bean Validation を作成する場合、カスタム制約には [制約アノテーションの作成](#) と [制約バリデーターの実装](#) が必要になります。以下のコードサンプルは、JBoss EAP に同梱される **bean-validation-custom-constraint** クイックスタートから抜粋したものです。完全な作業例はこのクイックスタートを参照してください。

13.2.3.1. 制約アノテーションの作成

以下は、**AddressValidator** クラスで定義されたカスタム制約のセットを使用して、エンティティー **Person** の **personAddress** フィールドがバリデーションされる例を表しています。

1. エンティティー **Person** を作成します。

例: Person クラス

```
package org.jboss.as.quickstarts.bean_validation_custom_constraint;

@Entity
@Table(name = "person")
public class Person implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
```

```

@Column(name = "person_id")
private Long personId;

@NotNull

@Size(min = 4)
private String firstName;

@NotNull
@Size(min = 4)
private String lastName;

// Custom Constraint @Address for bean validation
@NotNull
@Address
@OneToOne(mappedBy = "person", cascade = CascadeType.ALL)
private PersonAddress personAddress;

public Person() {

}

public Person(String firstName, String lastName, PersonAddress address) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.personAddress = address;
}

/* getters and setters omitted for brevity*/
}

```

2. 制約バリデーターファイルを作成します。

例: Address インターフェース

```

package org.jboss.as.quickstarts.bean_validation_custom_constraint;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;

// Linking the AddressValidator class with @Address annotation.
@Constraint(validatedBy = { AddressValidator.class })
// This constraint annotation can be used only on fields and method parameters.
@Target({ ElementType.FIELD, ElementType.PARAMETER })
@Retention(value = RetentionPolicy.RUNTIME)
@Documented
public @interface Address {

    // The message to return when the instance of MyAddress fails the validation.
    String message() default "Address Fields must not be null/empty and obey character limit constraints";
}

```

```

Class<?>[] groups() default {};

Class<? extends Payload>[] payload() default {};
}

```

例: PersonAddress クラス

```

package org.jboss.as.quickstarts.bean_validation_custom_constraint;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

@Entity
@Table(name = "person_address")
public class PersonAddress implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "person_id", unique = true, nullable = false)
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long personId;

    private String streetAddress;
    private String locality;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    @OneToOne
    @PrimaryKeyJoinColumn
    private Person person;

    public PersonAddress() {

    }

    public PersonAddress(String streetAddress, String locality, String city, String state, String
country, String pinCode) {
        this.streetAddress = streetAddress;
        this.locality = locality;
        this.city = city;
        this.state = state;
        this.country = country;
        this.pinCode = pinCode;
    }
}

```

```

    }
    /* getters and setters omitted for brevity */
}

```

13.2.3.2. 制約バリデーターの実装

アノテーションを定義したら、**@Address** アノテーションが付けられた要素のバリデーションが可能な制約バリデーターを作成する必要があります。これには、以下のように **ConstraintValidator** インターフェースを実装します。

例: AddressValidator クラス

```

package org.jboss.as.quickstarts.bean_validation_custom_constraint;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import org.jboss.as.quickstarts.bean_validation_custom_constraint.PersonAddress;

public class AddressValidator implements ConstraintValidator<Address, PersonAddress> {

    public void initialize(Address constraintAnnotation) {
    }

    /**
     * 1. A null address is handled by the @NotNull constraint on the @Address.
     * 2. The address should have all the data values specified.
     * 3. Pin code in the address should be of at least 6 characters.
     * 4. The country in the address should be of at least 4 characters.
     */

    public boolean isValid(PersonAddress value, ConstraintValidatorContext context) {
        if (value == null) {
            return true;
        }

        if (value.getCity() == null || value.getCountry() == null || value.getLocality() == null
            || value.getPinCode() == null || value.getState() == null || value.getStreetAddress() == null) {
            return false;
        }

        if (value.getCity().isEmpty()
            || value.getCountry().isEmpty() || value.getLocality().isEmpty()
            || value.getPinCode().isEmpty() || value.getState().isEmpty() ||
value.getStreetAddress().isEmpty()) {
            return false;
        }

        if (value.getPinCode().length() < 6) {
            return false;
        }

        if (value.getCountry().length() < 4) {
            return false;
        }
    }
}

```

```

    return true;
  }
}

```

13.3. バリデーション設定

Bean Validation は、**/META-INF** ディレクトリーにある **validation.xml** ファイル内の XML を使用して設定できます。このファイルがクラスパスに存在する場合は、**ValidatorFactory** が作成されたときに設定が適用されます。

例: バリデーション設定ファイル

以下の例は、**validation.xml** ファイルの複数の設定オプションを示しています。これらすべての設定はオプションです。これらのオプションは、**javax.validation** パッケージを使用して設定することもできます。

```

<validation-config xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration">

  <default-provider>
    org.hibernate.validator.HibernateValidator
  </default-provider>
  <message-interpolator>
    org.hibernate.validator.messageinterpolation.ResourceBundleMessageInterpolator
  </message-interpolator>
  <constraint-validator-factory>
    org.hibernate.validator.engine.ConstraintValidatorFactoryImpl
  </constraint-validator-factory>

  <constraint-mapping>
    /constraints-example.xml
  </constraint-mapping>

  <property name="prop1">value1</property>
  <property name="prop2">value2</property>
</validation-config>

```

ノード **default-provider** では、Bean Validation プロバイダーを選択できます。これは、クラスパスに複数のプロバイダーがある場合に役に立ちます。**message-interpolator** プロパティと **constraint-validator-factory** プロパティは、**javax.validation** パッケージで定義されたインターフェース **MessageInterpolator** および **ConstraintValidatorFactory** の使用済み実装をカスタマイズするために使用されます。**constraint-mapping** 要素は、実際の制約設定が含まれる追加の XML ファイルをリストします。

第14章 WEBSOCKET アプリケーションの作成

WebSocket プロトコルは、Web クライアントとサーバー間の双方向通信を提供します。クライアントとサーバー間の通信はイベントベースであるため、ポーリングベースの通信よりも処理が高速になり、帯域幅が小さくなります。WebSocket は、JavaScript API を用いて Web アプリケーションで使用したり、Java [WebSocket API](#) を用いてクライアント WebSocket エンドポイントで使用したりできます。

最初に接続はクライアントとサーバー間で HTTP 接続として確立されます。その後、クライアントは **Upgrade** ヘッダーを使用して WebSocket 接続を要求します。同じ TCP/IP 接続上ではすべて全二重通信になり、データのオーバーヘッドが最小化されます。各メッセージには不必要な HTTP ヘッダーコンテンツが含まれていないため、WebSocket 通信に必要な帯域幅は小さくなります。その結果、通信パスのレイテンシーが低くなるため、リアルタイムの応答が必要なアプリケーションに適しています。

JBoss EAP WebSocket 実装は、サーバーエンドポイントに対して完全な依存関係注入サポートを提供しますが、クライアントエンドポイントに対して CDI サービスを提供しません。

WebSocket アプリケーションには以下のコンポーネントと設定変更が必要です。

- Javaクライアントまたは WebSocket が有効になっている HTML クライアント。HTML クライアントのブラウザサポートは、<http://caniuse.com/#feat=websockets> を参照してください。
- WebSocket サーバーエンドポイントクラス。
- WebSocket API で依存関係を宣言するために設定されたプロジェクト依存関係。

WebSocket アプリケーションの作成

以下のコード例は、JBoss EAP に同梱される **websocket-hello** クイックスタートの一部です。これは、接続を開き、メッセージを送信し、接続を閉じる WebSocket アプリケーションの単純な例です。他の機能を実装したり、実際のアプリケーションで必要となるエラー処理を含むことはありません。

1. JavaScript HTML クライアントを作成します。
以下は WebSocket クライアントの例になります。この例には 3 つの JavaScript 関数が含まれています。
 - **connect()**: この関数は WebSocket URI を渡す WebSocket 接続を作成します。リソースの場所は、サーバーエンドポイントクラスに定義されたリソースと一致します。この関数は、WebSocket の **onopen**、**onmessage**、**onerror**、および **onclose** もインターセプトし、処理します。
 - **sendMessage()**: この関数はフォームに入力された名前を取得し、メッセージを作成します。さらに、WebSocket.send() コマンドを使用してメッセージを送信します。
 - **disconnect()**: この関数は WebSocket.close() コマンドを実行します。
 - **displayMessage()**: この関数は、ページ上の表示メッセージを WebSocket エンドポイントメソッドによって返された値に設定します。
 - **displayStatus()**: この関数は WebSocket の接続状態を表示します。

例: アプリケーション index.html コード

```
<html>
<head>
  <title>WebSocket: Say Hello</title>
  <link rel="stylesheet" type="text/css" href="resources/css/hello.css" />
  <script type="text/javascript">
```

```
var websocket = null;
function connect() {
    var wsURI = 'ws://' + window.location.host + '/websocket-
hello/websocket/helloName';
    websocket = new WebSocket(wsURI);
    websocket.onopen = function() {
        displayStatus('Open');
        document.getElementById('sayHello').disabled = false;
        displayMessage('Connection is now open. Type a name and click Say Hello to
send a message.');
```

};

```
    websocket.onmessage = function(event) {
        // log the event
        displayMessage('The response was received! ' + event.data, 'success');
    };
    websocket.onerror = function(event) {
        // log the event
        displayMessage('Error! ' + event.data, 'error');
    };
    websocket.onclose = function() {
        displayStatus('Closed');
        displayMessage('The connection was closed or timed out. Please click the Open
Connection button to reconnect.');
```

document.getElementById('sayHello').disabled = true;

```
    };
}
function disconnect() {
    if (websocket !== null) {
        websocket.close();
        websocket = null;
    }
    message.setAttribute("class", "message");
    message.value = 'WebSocket closed.';
    // log the event
}
function sendMessage() {
    if (websocket !== null) {
        var content = document.getElementById('name').value;
        websocket.send(content);
    } else {
        displayMessage('WebSocket connection is not established. Please click the Open
Connection button.', 'error');
```

};

```
    }
}
function displayMessage(data, style) {
    var message = document.getElementById('hellomessage');
    message.setAttribute("class", style);
    message.value = data;
}
function displayStatus(status) {
    var currentStatus = document.getElementById('currentstatus');
    currentStatus.value = status;
}
</script>
</head>
<body>
```

```

<div>
  <h1>Welcome to Red Hat JBoss Enterprise Application Platform!</h1>
  <div>This is a simple example of a WebSocket implementation.</div>
  <div id="connect-container">
    <div>
      <fieldset>
        <legend>Connect or disconnect using websocket :</legend>
        <input type="button" id="connect" onclick="connect();" value="Open Connection"
      />
        <input type="button" id="disconnect" onclick="disconnect();" value="Close
Connection" />
      </fieldset>
    </div>
    <div>
      <fieldset>
        <legend>Type your name below, then click the `Say Hello` button :</legend>
        <input id="name" type="text" size="40" style="width: 40%"/>
        <input type="button" id="sayHello" onclick="sendMessage();" value="Say Hello"
disabled="disabled"/>
      </fieldset>
    </div>
    <div>Current WebSocket Connection Status: <output id="currentstatus"
class="message">Closed</output></div>
    <div>
      <output id="hellomessage" />
    </div>
  </div>
</body>
</html>

```

2. WebSocket サーバーエンドポイントを作成します。

以下の方法のいずれかを使用して WebSocket サーバーエンドポイントを作成できます。

- **Programmatic Endpoint:** エンドポイントは Endpoint クラスを拡張します。
- **Annotated Endpoint:** エンドポイントクラスはアノテーションを使用して WebSocket イベントと対話します。これは、プログラマ的なエンドポイントよりも簡単にコーディングできます。

以下のコード例では、アノテーション付きエンドポイントが使用され、以下のイベントが処理されます。

- **@ServerEndpoint** アノテーションは、このクラスを WebSocket サーバーエンドポイントとして識別し、パスを指定します。
- WebSocket 接続が開かれると **@OnOpen** アノテーションがトリガーされます。
- メッセージが受信されると、**@OnMessage** アノテーションがトリガーされます。
- WebSocket 接続が閉じられると、**@OnClose** アノテーションがトリガーされます。

例: WebSocket エンドポイントコード

```
package org.jboss.as.quickstarts.websocket_hello;
```

```
import javax.websocket.CloseReason;
import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/websocket/helloName")
public class HelloName {

    @OnMessage
    public String sayHello(String name) {
        System.out.println("Say hello to " + name + "");
        return ("Hello" + name);
    }

    @OnOpen
    public void helloOnOpen(Session session) {
        System.out.println("WebSocket opened: " + session.getId());
    }

    @OnClose
    public void helloOnClose(CloseReason reason) {
        System.out.println("WebSocket connection closed with CloseCode: " +
reason.getCloseCode());
    }
}
```

3. プロジェクト POM ファイルで WebSocket API の依存関係を宣言します。
Maven を使用する場合は、プロジェクト **pom.xml** ファイルに以下の依存関係を追加します。

例: Maven 依存関係

```
<dependency>
  <groupId>org.jboss.spec.javax.websocket</groupId>
  <artifactId>jboss-websocket-api_1.1_spec</artifactId>
  <scope>provided</scope>
</dependency>
```

JBoss EAP に同梱されるクイックスタートには、追加の WebSocket クライアントとエンドポイントのコード例が含まれます。

第15章 JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS)

15.1. JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS)

JACC (Java Authorization Contract for Containers) はコンテナと承認サービスプロバイダー間のコントラクトを定義する規格であり、これによりコンテナによって使用されるプロバイダーの実装が可能になります。JACC は、Java コミュニティープロセスの JSR-115 で定義されています。この仕様の詳細は、「[Java™ Authorization Contract for Containers](#)」を参照してください。

JBoss EAP は、**security** サブシステムのセキュリティー機能内に JACC のサポートを実装します。

15.2. JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS) のセキュリティーの設定

JACC (Java Authorization Contract for Containers) を設定するには、適切なモジュールでセキュリティードメインを設定し、必須のパラメーターが含まれるよう **jboss-web.xml** を編集する必要があります。

セキュリティードメインへの JACC サポートの追加

セキュリティードメインに JACC サポートを追加するには、**required** フラグセットで **JACC** 承認ポリシーをセキュリティードメインの承認スタックへ追加します。以下は JACC サポートを持つセキュリティードメインの例です。ただし、セキュリティードメインは直接 XML を変更せずに、管理コンソールまたは管理 CLI で設定することが推奨されます。

例: JACC サポートのあるセキュリティードメイン

```
<security-domain name="jacc" cache-type="default">
  <authentication>
    <login-module code="UsersRoles" flag="required">
    </login-module>
  </authentication>
  <authorization>
    <policy-module code="JACC" flag="required"/>
  </authorization>
</security-domain>
```

JACC を使用するよう Web アプリケーションを設定

jboss-web.xml はデプロイメントの **WEB-INF/** ディレクトリに存在し、Web コンテナに対する追加の JBoss 固有の設定を格納し、上書きします。JACC が有効になっているセキュリティードメインを使用するには、**<security-domain>** 要素が含まれるようにし、さらに **<use-jboss-authorization>** 要素を **true** に設定する必要があります。以下の XML は、上記の JACC セキュリティードメインを使用するよう設定されています。

例: JACC セキュリティードメインの使用

```
<jboss-web>
  <security-domain>jacc</security-domain>
  <use-jboss-authorization>true</use-jboss-authorization>
</jboss-web>
```

JACC を使用するよう EJB アプリケーションを設定

セキュリティドメインと JACC を使用するよう EJB を設定する方法は Web アプリケーションの場合とは異なります。EJB の場合、**ejb-jar.xml** 記述子にてメソッドまたはメソッドのグループ上でメソッドパーミッションを宣言できます。**<ejb-jar>** 要素内では、すべての子 **<method-permission>** 要素に JACC ロールに関する情報が含まれます。詳細は、設定例を参照してください。

EJBMethodPermission クラスは Java EE API の一部であり、

https://access.redhat.com/webassets/avalon/d/red_hat_jboss_enterprise_application_platform/7.2/j で説明されています。

例: EJB の JACC メソッドパーミッション

```
<ejb-jar>
  <assembly-descriptor>
    <method-permission>
      <description>The employee and temp-employee roles can access any method of the
EmployeeService bean </description>
      <role-name>employee</role-name>
      <role-name>temp-employee</role-name>
      <method>
        <ejb-name>EmployeeService</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>
  </assembly-descriptor>
</ejb-jar>
```

Web アプリケーションと同様にセキュリティドメインを使用して EJB の認証および承認メカニズムを指定することも可能です。セキュリティドメインは **<security>** 子要素の **jboss-ejb3.xml** 記述子に宣言されます。セキュリティドメインの他に、EJB が実行されるプリンシパルを変更する **<run-as-principal>** を指定することもできます。

例: EJB のセキュリティドメイン宣言

```
<ejb-jar>
  <assembly-descriptor>
    <security>
      <ejb-name>*</ejb-name>
      <security-domain>myDomain</security-domain>
      <run-as-principal>myPrincipal</run-as-principal>
    </security>
  </assembly-descriptor>
</ejb-jar>
```

elytron サブシステムを使用した JACC の有効化

レガシーセキュリティーサブシステムでの JACC の無効化

アプリケーションサーバーはデフォルトでレガシー **security** サブシステムを使用して、JACC ポリシープロバイダーおよびファクトリーを設定します。デフォルト設定は PicketBox から実装へマップします。

Elytron またはアプリケーションサーバーにインストールするその他のポリシーを使用して JACC 設定を管理するには、最初にレガシー **security** サブシステムで JACC を無効にする必要があります。これには、以下の管理 CLI コマンドを使用できます。

■

```
/subsystem=security:write-attribute(name=initialize-jacc, value=false)
```

この作業を怠ると、次のエラーメッセージがサーバーログに出力されます: **MSC000004: Failure during stop of service org.wildfly.security.policy: java.lang.StackOverflowError**

JACC ポリシープロバイダーの定義

elytron サブシステムは、JACC 仕様を基にして組み込みのポリシープロバイダーを提供します。ポリシープロバイダーを作成するには、以下の管理 CLI コマンドを実行します。

```
/subsystem=elytron/policy=jacc:add(jacc-policy={})
reload
```

Web デプロイメントに対する JACC の有効化

JACC ポリシープロバイダーを定義したら、以下のコマンドを実行して、web デプロイメントに対して JACC を有効にできます。

```
/subsystem=undertow/application-security-domain=other:add(security-domain=ApplicationDomain,enable-jacc=true)
```

上記のコマンドは、**jboss-web.xml** ファイルに指定がない場合に、アプリケーションのデフォルトのセキュリティドメインを定義します。すでに **application-security-domain** が定義されている場合、以下のコマンドを実行すると JACC を有効にすることができます。

```
/subsystem=undertow/application-security-domain=my-security-domain:write-attribute(name=enable-jacc,value=true)
```

EJB デプロイメントに対する JACC の有効化

JACC ポリシープロバイダーを定義したら、以下のコマンドを実行して、EJB デプロイメントに対して JACC を有効にできます。

```
/subsystem=ejb3/application-security-domain=other:add(security-domain=ApplicationDomain,enable-jacc=true)
```

上記のコマンドは、EJB のデフォルトのセキュリティドメインを定義します。すでに **application-security-domain** が定義されている場合、以下のコマンドを実行すると JACC を有効にすることができます。

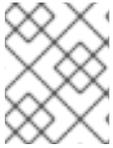
```
/subsystem=ejb3/application-security-domain=my-security-domain:write-attribute(name=enable-jacc,value=true)
```

カスタム Elytron ポリシープロバイダーの作成

パーミッションをチェックするために一部の外部承認サービスと統合したい場合など、カスタム **java.security.Policy** が必要なときにカスタムのポリシープロバイダーが使用されます。カスタムポリシープロバイダーを作成するには、**java.security.Policy** を実装し、実装でカスタムモジュールを作成およびプラグし、**elytron** サブシステムのモジュールから実装を使用します。

```
/subsystem=elytron/policy=policy-provider-a:add(custom-policy={class-name=MyPolicyProviderA,module=x.y.z})
```

詳細は「[ポリシープロバイダープロパティ](#)」を参照してください。



注記

ほとんどの場合で、JACC ポリシープロバイダーを Java EE 対応のアプリケーションサーバーの一部として想定どおりに使用できます。

第16章 JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS)

16.1. JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS) のセキュリティ

Java Authentication SPI for Containers (JASPI または JASPIC) は Java アプリケーションのプラグ可能なインターフェースであり、Java Community Process の JSR-196 に定義されています。JACC は、Java コミュニティプロセスの JSR-196 で定義されています。この仕様の詳細は、<http://www.jcp.org/en/jsr/detail?id=196> を参照してください。

16.2. JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS) のセキュリティの設定

JASPI プロバイダーを認証するには、`<authentication-jaspi>` 要素をセキュリティドメインに追加します。設定は標準的な認証モジュールと似ていますが、ログインモジュール要素は `<login-module-stack>` 要素で囲まれています。設定の構成は次のとおりです。

例: authentication-jaspi 要素の構造

```
<authentication-jaspi>
  <login-module-stack name="...">
    <login-module code="..." flag="...">
      <module-option name="..." value="..." />
    </login-module>
  </login-module-stack>
  <auth-module code="..." login-module-stack-ref="...">
    <module-option name="..." value="..." />
  </auth-module>
</authentication-jaspi>
```

ログインモジュール自体は標準的な認証モジュールと同じように設定されます。

Web ベースの管理コンソールは JASPI 認証モジュールの設定を公開しません。そのため、JBoss EAP の実行中のインスタンスを完全に停止してから、設定を

`EAP_HOME/domain/configuration/domain.xml` ファイルまたは

`EAP_HOME/standalone/configuration/standalone.xml` ファイルに直接追加する必要があります。

第17章 JAVA バッチアプリケーション開発

JBoss EAP 7以降、JBoss EAP では [JSR-352](#) で定義された Java バッチアプリケーションがサポートされます。JBoss EAP の **batch-jberet** サブシステムにより、バッチ設定と監視が行えるようになります。

JBoss EAP でバッチ処理を使用するようアプリケーションを設定するには、[必要な依存関係](#)を指定する必要があります。バッチ処理向けの追加の JBoss EAP 機能には、[Job Specification Language \(JSL\) 継承](#)と[バッチプロパティインジェクション](#)が含まれます。

17.1. 必要なバッチ依存関係

JBoss EAP にバッチアプリケーションをデプロイするには、バッチ処理に必要な追加の依存関係をアプリケーションの **pom.xml** で宣言する必要があります。必要なこれらの依存関係の例を以下に示します。ほとんどの依存関係は JBoss EAP にすでに含まれているため、スコープは **provided** に設定されます。

例: pom.xml バッチ依存関係

```
<dependencies>
  <dependency>
    <groupId>org.jboss.spec.java.batch</groupId>
    <artifactId>jboss-batch-api_1.0_spec</artifactId>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>javax.enterprise</groupId>
    <artifactId>cdi-api</artifactId>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>org.jboss.spec.java.annotation</groupId>
    <artifactId>jboss-annotations-api_1.2_spec</artifactId>
    <scope>provided</scope>
  </dependency>

  <!-- Include your application's other dependencies. -->
  ...
</dependencies>
```

17.2. JOB SPECIFICATION LANGUAGE (JSL) 継承

JBoss EAP **batch-jberet** サブシステムの機能を使用すると、Job Specification Language (JSL) 継承を使用してジョブ定義の共通の部分を実装化できます。JSL 継承は JSR-352 1.0 仕様に含まれていないため、JBoss EAP **batch-jberet** サブシステムは JSL Inheritance v1 のドラフト版に基づいて JSL 継承を実装します。

同じジョブ XML ファイル内の step および flow の継承

step や flow などの親要素は、直接的な実行から除外するために属性 **abstract="true"** でマークされます。子要素には、親要素を参照する **parent** 属性が含まれます。

```
<job id="inheritance" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
```

```

<!-- abstract step and flow -->
<step id="step0" abstract="true">
  <batchlet ref="batchlet0"/>
</step>

<flow id="flow0" abstract="true">
  <step id="flow0.step1" parent="step0"/>
</flow>

<!-- concrete step and flow -->
<step id="step1" parent="step0" next="flow1"/>

<flow id="flow1" parent="flow0"/>
</job>

```

異なるジョブ XML ファイルからのステップの継承

step や job などの子要素には以下が含まれます。

- **jsl-name** 属性。親要素を含むジョブ XML ファイルの名前 (.xml 拡張子なし) を指定します。
- **parent** 属性。**jsl-name** で指定されたジョブ XML ファイルの親要素を参照します。

親要素は、直接的な実行から除外するために属性 **abstract="true"** でマークされます。

例: chunk-child.xml

```

<job id="chunk-child" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
  <step id="chunk-child-step" parent="chunk-parent-step" jsl-name="chunk-parent">
  </step>
</job>

```

例: chunk-parent.xml

```

<job id="chunk-parent" >
  <step id="chunk-parent-step" abstract="true">
    <chunk checkpoint-policy="item" skip-limit="5" retry-limit="5">
      <reader ref="R1"></reader>
      <processor ref="P1"></processor>
      <writer ref="W1"></writer>

      <checkpoint-algorithm ref="parent">
        <properties>
          <property name="parent" value="parent"></property>
        </properties>
      </checkpoint-algorithm>
      <skippable-exception-classes>
        <include class="java.lang.Exception"></include>
        <exclude class="java.io.IOException"></exclude>
      </skippable-exception-classes>
      <retryable-exception-classes>
        <include class="java.lang.Exception"></include>
        <exclude class="java.io.IOException"></exclude>
      </retryable-exception-classes>
      <no-rollback-exception-classes>
        <include class="java.lang.Exception"></include>

```

```
<exclude class="java.io.IOException"></exclude>
</no-rollback-exception-classes>
</chunk>
</step>
</job>
```

17.3. バッチプロパティインジェクション

JBoss EAP **batch-jberet** サブシステムの機能を使用すると、ジョブ XML ファイルで定義されたプロパティをバッチアーティファクトクラスのフィールドにインジェクトできます。ジョブ XML ファイルで定義されたプロパティは **@Inject** アノテーションと **@BatchProperty** アノテーションを使用してフィールドにインジェクトできます。

インジェクトフィールドは以下のいずれかの Java タイプになります。

- **java.lang.String**
- **java.lang.StringBuilder**
- **java.lang.StringBuffer**
- 以下のいずれかのプリミティブタイプおよびラッパータイプ:
 - **boolean**、**Boolean**
 - **int**、**Integer**
 - **double**、**Double**
 - **long**、**Long**
 - **char**、**Character**
 - **float**、**Float**
 - **short**、**Short**
 - **byte**、**Byte**
- **java.math.BigInteger**
- **java.math.BigDecimal**
- **java.net.URL**
- **java.net.URI**
- **java.io.File**
- **java.util.jar.JarFile**
- **java.util.Date**
- **java.lang.Class**
- **java.net.Inet4Address**

- `java.net.Inet6Address`
- `java.util.List`、`List<?>`、`List<String>`
- `java.util.Set`、`Set<?>`、`Set<String>`
- `java.util.Map`、`Map<?, ?>`、`Map<String, String>`、`Map<String, ?>`
- `java.util.logging.Logger`
- `java.util.regex.Pattern`
- `javax.management.ObjectName`

以下のアレイタイプもサポートされています。

- `java.lang.String[]`
- 以下のいずれかのプリミティブタイプおよびラッパータイプ:
 - `boolean[]`、`Boolean[]`
 - `int[]`、`Integer[]`
 - `double[]`、`Double[]`
 - `long[]`、`Long[]`
 - `char[]`、`Character[]`
 - `float[]`、`Float[]`
 - `short[]`、`Short[]`
 - `byte[]`、`Byte[]`
- `java.math.BigInteger[]`
- `java.math.BigDecimal[]`
- `java.net.URL[]`
- `java.net.URI[]`
- `java.io.File[]`
- `java.util.jar.JarFile[]`
- `java.util.zip.ZipFile[]`
- `java.util.Date[]`
- `java.lang.Class[]`

以下に、バッチプロパティインジェクションの使用例をいくつか示します。

- [数字を Batchlet クラスにさまざまなタイプとしてインジェクトする](#)

- 数字シーケンスを Batchlet クラスにさまざまなアレイとしてインジェクトする
- Batchlet クラスにクラスプロパティをインジェクトする
- プロパティインジェクションに対してアノテーション付けされたフィールドにデフォルト値を割り当てる

数字を Batchlet クラスにさまざまなタイプとしてインジェクトする

例: Job XML ファイル

```
<batchlet ref="myBatchlet">
  <properties>
    <property name="number" value="10"/>
  </properties>
</batchlet>
```

例: アーティファクトクラス

```
@Named
public class MyBatchlet extends AbstractBatchlet {
    @Inject
    @BatchProperty
    int number; // Field name is the same as batch property name.

    @Inject
    @BatchProperty (name = "number") // Use the name attribute to locate the batch property.
    long asLong; // Inject it as a specific data type.

    @Inject
    @BatchProperty (name = "number")
    Double asDouble;

    @Inject
    @BatchProperty (name = "number")
    private String asString;

    @Inject
    @BatchProperty (name = "number")
    BigInteger asBigInteger;

    @Inject
    @BatchProperty (name = "number")
    BigDecimal asBigDecimal;
}
```

数字シーケンスを Batchlet クラスにさまざまなアレイとしてインジェクトする

例: Job XML ファイル

```
<batchlet ref="myBatchlet">
  <properties>
    <property name="weekDays" value="1,2,3,4,5,6,7"/>
  </properties>
</batchlet>
```

例: アーティファクトクラス

```

@Named
public class MyBatchlet extends AbstractBatchlet {
    @Inject
    @BatchProperty
    int[] weekDays; // Array name is the same as batch property name.

    @Inject
    @BatchProperty (name = "weekDays") // Use the name attribute to locate the batch property.
    Integer[] asIntegers; // Inject it as a specific array type.

    @Inject
    @BatchProperty (name = "weekDays")
    String[] asStrings;

    @Inject
    @BatchProperty (name = "weekDays")
    byte[] asBytes;

    @Inject
    @BatchProperty (name = "weekDays")
    BigInteger[] asBigIntegers;

    @Inject
    @BatchProperty (name = "weekDays")
    BigDecimal[] asBigDecimals;

    @Inject
    @BatchProperty (name = "weekDays")
    List asList;

    @Inject
    @BatchProperty (name = "weekDays")
    List<String> asListString;

    @Inject
    @BatchProperty (name = "weekDays")
    Set asSet;

    @Inject
    @BatchProperty (name = "weekDays")
    Set<String> asSetString;
}

```

Batchlet クラスにクラスプロパティをインジェクトする

例: Job XML ファイル

```

<batchlet ref="myBatchlet">
  <properties>
    <property name="myClass" value="org.jberet.support.io.Person"/>
  </properties>
</batchlet>

```

例: アーティファクトクラス

```
@Named
public class MyBatchlet extends AbstractBatchlet {
    @Inject
    @BatchProperty
    private Class myClass;
}
```

プロパティインジェクションに対してアノテーション付けされたフィールドにデフォルト値を割り当てる

ターゲットバッチプロパティがジョブXMLファイルで定義されていない場合は、アーティファクトJavaクラスのフィールドにデフォルト値を割り当てることができます。ターゲットプロパティが有効な値に解決される場合は、その値がそのフィールドにインジェクトされます。解決されない場合は、値がインジェクトされず、デフォルトのフィールド値が使用されます。

例: アーティファクトクラス

```
/**
 * Comment character. If commentChar batch property is not specified in job XML file, use the default
 * value '#'.
 */
@Inject
@BatchProperty
private char commentChar = '#';
```

第18章 クライアントの設定

18.1. WILDFLY-CONFIG.XML ファイルを使用したクライアント設定

EJB や naming などの JBoss EAP クライアントライブラリーは、リリース 7.1 まで異なる設定ストラテジーを使用していました。サーバー設定と似た方法で、すべてのクライアント設定を1つの設定ファイルに統合するため、JBoss EAP 7.1 には **wildfly-config.xml** ファイルが導入されました。

以下の表は、JBoss EAP の **wildfly-config.xml** ファイルを使用して実現できるクライアント設定および設定タイプと、リファレンススキーマリンクを示しています。

クライアント設定	スキーマの場所 / 設定情報
認証クライアント	<p>スキーマリファレンスは、EAP_HOME/docs/schema/elytron-client-1_2.xsd の製品インストールに提供されます。</p> <p>スキーマは、http://www.jboss.org/schema/jbossas/elytron-client-1_2.xsd にも公開されています。</p> <p>詳細と設定例は、「wildfly-config.xml ファイルを使用したクライアント認証設定」を参照してください。</p> <p>詳細は、JBoss EAP 『How to Configure Identity Management』の「Configure Client Authentication with Elytron Client」を参照してください。</p>
EJB クライアント	<p>スキーマリファレンスは、EAP_HOME/docs/schema/wildfly-client-ejb_3_0.xsd の製品インストールに提供されます。</p> <p>スキーマは http://www.jboss.org/schema/jbossas/wildfly-client-ejb_3_0.xsd にも公開されます。</p> <p>詳細と設定例は、「wildfly-config.xml ファイルを使用した EJB クライアント設定」を参照してください。</p> <p>その他の簡単な例は、JBoss EAP 『移行ガイド』の「EJB クライアントの Elytron への移行」を参照してください。</p>
HTTP クライアント	<p>スキーマリファレンスは、EAP_HOME/docs/schema/wildfly-http-client_1_0.xsd の製品インストールに提供されます。</p> <p>スキーマは http://www.jboss.org/schema/jbossas/wildfly-http-client_1_0.xsd にも公開されます。</p> <div style="display: flex; align-items: flex-start; margin-top: 20px;"> <div style="flex: 1;">  </div> <div style="flex: 2;"> <p>注記</p> <p>この機能はテクノロジープレビューとしてのみ提供されません。</p> </div> </div> <p>詳細と設定例は、「wildfly-config.xml ファイルを使用した HTTP クライアント設定」を参照してください。</p>

クライアント設定	スキーマの場所 / 設定情報
リモートクライアント	<p>スキーマリファレンスは EAP_HOME/docs/schema/jboss-remoting_5_0.xsd の製品インストールに提供されます。</p> <p>スキーマは、http://www.jboss.org/schema/jbossas/jboss-remoting_5_0.xsd にも公開されています。</p> <p>詳細と設定例は、「wildfly-config.xml ファイルを使用したリモートクライアント設定」を参照してください。</p>
XNIO ワーカークライアント	<p>スキーマリファレンスは、EAP_HOME/docs/schema/xnio_3_5.xsd の製品インストールに提供されます。</p> <p>スキーマは、http://www.jboss.org/schema/jbossas/xnio_3_5.xsd にも公開されています。</p> <p>詳細と設定例は、「wildfly-config.xml ファイルを使用したデフォルトの XNIO ワーカー設定」を参照してください。</p>

18.1.1. wildfly-config.xml ファイルを使用したクライアント認証設定

urn:elytron:client:1.2 ネームスペースにある **authentication-client** 要素を使用すると、**wildfly-config.xml** ファイルを使用してクライアント認証を設定できます。ここでは、この要素を使用したクライアント認証の設定方法について説明します。

authentication-client 要素および属性

authentication-client 要素には、任意で以下のトップレベル子要素とそれらの子要素を含むことが可能です。

- **credential-stores**
 - **credential-store**
 - **providers**
 - **global**
 - **use-service-loader**
 - **attributes**
 - **protection-parameter-credentials**
 - **key-store-reference**
 - **credential-store-reference**
 - **clear-password**
 - **key-pair**
 - **public-key-pem**
 - **private-key-pem**

- **certificate**
 - **public-key-pem**
- **bearer-token**
- **oauth2-bearer-token**
 - **client-credentials**
 - **resource-owner-credentials**
- **key-stores**
 - **key-store**
 - **file**
 - **load-from**
 - **resource**
 - **key-store-clear-password**
 - **key-store-credential**
- **authentication-rules**
 - **rule**
 - **match-no-user**
 - **match-user**
 - **match-protocol**
 - **match-host**
 - **match-path**
 - **match-port**
 - **match-urn**
 - **match-domain-name**
 - **match-abstract-type**
- **authentication-configurations**
 - **configuration**
 - **set-host-name**
 - **set-port-number**
 - **set-protocol**
 - **set-user-name**

- **set-anonymous**
- **set-mechanism-realm-name**
- **rewrite-user-name-regex**
- **sasl-mechanism-selector**
- **set-mechanism-properties**
 - **property**
- **credentials**
 - **key-store-reference**
 - **credential-store-reference**
 - **clear-password**
 - **key-pair**
 - **certificate**
 - **public-key-pem**
 - **bearer-token**
 - **oauth2-bearer-token**
- **set-authorization-name**
- **providers**
 - **global**
 - **use-service-loader**
- **use-provider-sasl-factory**
- **use-service-loader-sasl-factory**
- **net-authenticator**
- **ssl-context-rules**
 - **rule**
 - **match-no-user**
 - **match-user**
 - **match-protocol**
 - **match-host**
 - **match-path**
 - **match-port**

- [match-urn](#)
 - [match-domain-name](#)
 - [match-abstract-type](#)
- [ssl-contexts](#)
 - [default-ssl-context](#)
 - [ssl-context](#)
 - [key-store-ssl-certificate](#)
 - [trust-store](#)
 - [cipher-suite](#)
 - [protocol](#)
 - [provider-name](#)
 - [certificate-revocation-list](#)
 - [providers](#)
 - [global](#)
 - [use-service-loader](#)
- [providers](#)
 - [global](#)
 - [use-service-loader](#)

credential-stores

この任意の要素は、設定内でクレデンシャルを埋め込む代わりに、設定内の他の場所から参照されるクレデンシャルストアを定義します。

任意数の [credential-store](#) 要素を含むことができます。

例: credential-stores 設定

```
<configuration>
<authentication-client xmlns="urn:elytron:client:1.2">
  <credential-stores>
    <credential-store name="..." type="..." provider="..." >
      <attributes>
        <attribute name="..." value="..." />
      </attributes>
      <protection-parameter-credentials>...</protection-parameter-credentials>
    </credential-store>
  </credential-stores>
</authentication-client>
</configuration>
```

credential-store

この要素は、設定の別の場所から参照されるクレデンシャルストアを定義します。
この要素は以下の属性を持ちます。

属性名	属性の説明
name	クレデンシャルストアの名前。必須の属性です。
type	クレデンシャルストアのタイプ。この属性は任意です。
provider	クレデンシャルストアのロードに使用する java.security.Provider の名前。 この属性は任意です。

この要素には、以下の各子要素を1つのみ含めることができます。

- [providers](#)
- [attributes](#)
- [protection-parameter-credentials](#)

attributes

この要素は、クレデンシャルストアの初期化に使用される設定属性を定義し、設定に必要な回数繰り返すことができます。

例: attributes 設定

```
<attributes>
  <attribute name="..." value="..." />
</attributes>
```

protection-parameter-credentials

この要素には、クレデンシャルストアを初期化するとき使用される保護パラメーターにアセンブルされる1つまたは複数のクレデンシャルを含むことができます。

クレデンシャルストアの実装に依存する以下の子要素を1つ以上含めることができます。

- [key-store-reference](#)
- [credential-store-reference](#)
- [clear-password](#)
- [key-pair](#)
- [certificate](#)
- [public-key-pem](#)
- [bearer-token](#)
- [oauth2-bearer-token](#)

例: protection-parameter-credentials 設定

```

<protection-parameter-credentials>
  <key-store-reference>...</key-store-reference>
  <credential-store-reference store="..." alias="..." clear-text="..." />
  <clear-password password="..." />
  <key-pair public-key-pem="..." private-key-pem="..." />
  <certificate private-key-pem="..." pem="..." />
  <public-key-pem>...</public-key-pem>
  <bearer-token value="..." />
  <oauth2-bearer-token token-endpoint-uri="...">...</oauth2-bearer-token>
</protection-parameter-credentials>

```

key-store-reference

この要素は、JBoss EAP の認証メカニズムによって [現在使用されていません](#)。キーストアへの参照を定義します。

この要素は以下の属性を持ちます。

属性名	属性の説明
key-store-name	キーストア名。必須の属性です。
alias	参照されたキーストアからロードするエントリーのエイリアス。単一のエントリーのみが含まれるキーストアのみ省略できます。

これには、以下の子要素の1つのみを含めることができます。

- [key-store-clear-password](#)
- [credential-store-reference](#)
- [key-store-credential](#)

例: key-store-reference 設定

```

<key-store-reference key-store-name="..." alias="...">
  <key-store-clear-password password="..." />
  <key-store-credential>...</key-store-credential>
</key-store-reference>

```

credential-store-reference

この要素はクレデンシャルストアへの参照を定義します。

この要素は以下の属性を持ちます。

属性名	属性の説明
store	クレデンシャルストア名。

属性名	属性の説明
alias	参照されたクレデンシャルストアからロードするエントリーのエイリアス。単一のエントリーのみが含まれるキーストアのみ省略できます。
clear-text	クリアテキストのパスワード。

clear-password

この要素は、クリアテキストパスワードを定義します。

key-pair

この要素は、JBoss EAP の認証メカニズムによって [現在使用されていません](#)。パブリックキーとプライベートキーのペアを定義します。

これには、以下の子要素を含めることができます。

- [public-key-pem](#)
- [private-key-pem](#)

public-key-pem

この要素は、JBoss EAP の認証メカニズムによって [現在使用されていません](#)。EPM エンコードのパブリックキーを定義します。

private-key-pem

この要素は、PEM エンコードのプライベートキーを定義します。

certificate

この要素は、JBoss EAP の認証メカニズムによって [現在使用されていません](#)。証明書を指定します。

この要素は以下の属性を持ちます。

属性名	属性の説明
private-key-pem	PEM エンコードのプライベートキー。
pem	対応する証明書。

bearer-token

この要素はベアラートークンを定義します。

oauth2-bearer-token

この要素は、OAuth 2 のベアラートークンを定義します。

この要素には以下の属性があります。

属性名	属性の説明
token-endpoint-uri	トークンエンドポイントの URL。

この要素には、以下の各子要素を1つのみ含めることができます。

- [client-credentials](#)
- [resource-owner-credentials](#)

client-credentials

この要素は、クライアントクレデンシャルを定義します。

この要素は以下の属性を持ちます。

属性名	属性の説明
client-id	クライアント ID。必須の属性です。
client-secret	クライアントシークレット。必須の属性です。

resource-owner-credentials

この要素は、リソース所有者のクレデンシャルを定義します。

この要素は以下の属性を持ちます。

属性名	属性の説明
name	リソース名。必須の属性です。
password	パスワード。必須の属性です。

key-stores

この要素は任意で、設定の別の場所から参照されるキーストアを定義します。

例: key-stores 設定

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <key-stores>
      <key-store name="...">
        <!-- The following 3 elements specify where to load the keystore from. -->
        <file name="..." />
        <load-from uri="..." />
        <resource name="..." />
        <!-- One of the following to specify the protection parameter to unlock the keystore. -->
        <key-store-clear-password password="..." />
      </key-store>
    </key-stores>
  </authentication-client>
</configuration>
```

```

    <key-store-credential>...</key-store-credential>
  </key-store>
</key-stores>
...
</authentication-client>
</configuration>

```

key-store

この要素は任意で、設定の他の場所から参照されるキーストアを定義します。

key-store には以下の属性があります。

属性名	属性の説明
name	キーストアの名前。必須の属性です。
type	JCEKS などのキーストアタイプ。必須の属性です。
provider	クレデンシャルストアのロードに使用する java.security.Provider の名前。この属性は任意です。
wrap-passwords	true の場合、 passwords をラップします。パスワードは、クリアなパスワードコンテンツを UTF-8 でエンコードした後、結果のバイトを秘密鍵として格納します。デフォルトは false です。

キーストアのロード元を定義する以下の要素の1つが必ず含まれる必要があります。

- [file](#)
- [load-from](#)
- [resource](#)

また、キーストアを初期化するとき使用する保護パラメーターを指定する以下の要素の1つが含まれている必要があります。

- [key-store-clear-password](#)
- [key-store-credential](#)

file

この要素はキーストアファイルの名前を指定します。

この要素には以下の属性があります。

属性名	属性の説明
name	ファイルの完全修飾ファイルパスおよび名前。

load-from

この要素は、キーストアファイルの URI を指定します。
この要素には以下の属性があります。

属性名	属性の説明
uri	キーストアファイルの URI。

resource

この要素は、**Thread** コンテキストクラスローダーからロードするリソースの名前を指定します。
この要素には以下の属性があります。

属性名	属性の説明
name	リソースの名前。

key-store-clear-password

この要素はクリアテキストパスワードを指定します。
この要素には以下の属性があります。

属性名	属性の説明
password	クリアテキストのパスワード。

key-store-credential

この要素は、このキーストアにアクセスするために保護パラメーターとして使用するエントリーを取得する別のキーストアへの参照を指定します。

key-store-credential 要素には以下の属性があります。

属性名	属性の説明
key-store-name	キーストア名。必須の属性です。
alias	参照されたキーストアからロードするエントリーのエイリアス。単一のエントリーのみが含まれるキーストアのみ省略できます。

これには、以下の子要素の1つのみを含めることができます。

- [key-store-clear-password](#)
- [credential-store-reference](#)
- [key-store-credential](#)

例: **key-store-credential** 設定

```

<key-store-credential key-store-name="..." alias="...">
  <key-store-clear-password password="..." />
</key-store-credential>...</key-store-credential>
</key-store-credential>

```

authentication-rules

この要素は、適切な認証設定を適用するためにアウトバウンド接続と照合するルールを定義します。**authentication-configuration** が必要である場合、アクセスされたリソースの URI と、任意の抽象型および抽象型オーソリティーは、設定に定義されたルールと照合され、使用する **authentication-configuration** を特定します。

この要素には、1つまたは複数の子 **rule** 要素を含めることができます。

例: authentication-rules 設定

```

<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    ...
    <authentication-rules>
      <rule use-configuration="...">
        ...
      </rule>
    </authentication-rules>
    ...
  </authentication-client>
</configuration>

```

rule

この要素は、適切な認証設定を適用するためにアウトバウンド接続と照合するルールを定義します。

この要素には以下の属性があります。

属性名	属性の説明
use-configuration	ルールが一致した時に選択される認証設定。

認証設定ルールの照合は、SSL コンテキストルールの照合とは独立しています。認証設定ルールは認証設定、SSL コンテキストルールは SSL コンテキストを参照すること以外は、認証ルールの構造は SSL コンテキストルールの構造と同じです。

これには、以下の子要素を含めることができます。

- **match-no-user**
- **match-user**
- **match-protocol**
- **match-host**
- **match-path**

- [match-port](#)
- [match-urn](#)
- [match-domain-name](#)
- [match-abstract-type](#)

例: 認証の rule 設定

```
<rule use-configuration="...">
  <!-- At most one of the following two can be defined. -->
  <match-no-user />
  <match-user name="..." />
  <!-- Each of the following can be defined at most once. -->
  <match-protocol name="..." />
  <match-host name="..." />
  <match-path name="..." />
  <match-port number="..." />
  <match-urn name="..." />
  <match-domain name="..." />
  <match-abstract-type name="..." authority="..." />
</rule>
```

match-no-user

URI 内に埋め込まれた **user-info** がない場合、このルールと一致します。

match-user

URI 内に埋め込まれた **user-info** がこの要素に指定された **name** 属性と一致する場合、このルールと一致します。

match-protocol

URI 内のプロトコルがこの要素に指定されたプロトコル **name** 属性と一致する場合、このルールと一致します。

match-host

URI 内に指定されたホスト名がこの要素に指定されたホスト **name** 属性と一致する場合、このルールと一致します。

match-path

URI 内に指定されたパスがこの要素に指定されたパス **name** 属性と一致する場合、このルールと一致します。

match-port

URI 内に指定されたポート番号がこの要素に指定されたポート **number** 属性と一致する場合、このルールと一致します。これは、URI 内に指定された番号のみと照合され、このプロトコルから派生するデフォルトのポート番号とは照合されません。

match-urn

URI のスキーム固有の部分がこの要素に指定された **name** 属性と一致する場合、このルールと一致します。

match-domain-name

URI のプロトコルが **domain** で、URI のスキーム固有の部分がこの要素に指定された **name** 属性と一致する場合、このルールと一致します。

match-abstract-type

抽象型が **name** 属性と一致し、オーソリティーがこの要素で指定された **authority** 属性と一致する場合、このルールと一致します。

authentication-configurations

この要素は、認証ルールによって選択される名前付きの認証設定を定義します。これには、1つまたは複数の **configuration** 要素を含めることができます。

例: authentication-configurations 設定

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <authentication-configurations>
      <configuration name="...">
        <!-- Destination Overrides. -->
        <set-host name="..." />
        <set-port number="..." />
        <set-protocol name="..." />
        <!-- At most one of the following two elements. -->
        <set-user-name name="..." />
        <set-anonymous />
        <set-mechanism-realm name="..." />
        <rewrite-user-name-regex pattern="..." replacement="..." />
        <sasl-mechanism-selector selector="..." />
        <set-mechanism-properties>
          <property key="..." value="..." />
        </set-mechanism-properties>
        <credentials>...</credentials>
        <set-authorization-name name="..." />
        <providers>...</providers>
        <!-- At most one of the following two elements. -->
        <use-provider-sasl-factory />
        <use-service-loader-sasl-factory module-name="..." />
      </configuration>
    </authentication-configurations>
  </authentication-client>
</configuration>
```

configuration

この要素は、認証ルールによって選択される名前付きの認証設定を定義します。これには、以下の子要素を含めることができます。

- 任意の **set-host-name**、**set-port-number**、および **set-protocol** 要素は宛先をオーバーライドできます。
- 任意の **set-user-name** および **set-anonymous** 要素は相互排他的で、認証の名前の設定や匿名認証への切り替えに使用できます。
- 次は任意の **set-mechanism-realm-name**、**rewrite-user-name-regex**、**sasl-mechanism-selector**、**set-mechanism-properties**、**credentials**、**set-authorization-name**、および **providers** 要素です。

- 最後の2つの要素 **use-provider-sasl-factory** および **use-service-loader-sasl-factory** は任意で、相互排他的です。SASL メカニズムファクトリーが認証に対して検出される方法を定義します。

set-host-name

この要素は、認証された呼び出しのホスト名をオーバーライドします。
この要素には以下の属性があります。

属性名	属性の説明
name	ホスト名。

set-port-number

この要素は、認証された呼び出しのポート番号をオーバーライドします。
この要素には以下の属性があります。

属性名	属性の説明
number	ポート番号。

set-protocol

この要素は、認証された呼び出しのプロトコルをオーバーライドします。
この要素には以下の属性があります。

属性名	属性の説明
name	プロトコル。

set-user-name

この要素は、認証に使用するユーザー名を設定します。 **set-anonymous** 要素とは使用しないでください。
この要素には以下の属性があります。

属性名	属性の説明
name	認証に使用するユーザー名。

set-anonymous

匿名認証への切り替えに使用されます。 **set-user-name** 要素とは使用しないでください。

set-mechanism-realm-name

この要素は、必要時に SASL メカニズムによって選択されるレルムの名前を指定します。
この要素には以下の属性があります。

属性名	属性の説明
name	レルムの名前。

rewrite-user-name-regex

この要素は、認証に使用されるユーザー名を再書き込みする正規表現パターンと代替を定義します。
この要素は以下の属性を持ちます。

属性名	属性の説明
pattern	正規表現パターン。
replacement	認証に使用されるユーザー名の再書き込みに使用する代替。

sasl-mechanism-selector

この要素は、[org.wildfly.security.sasl.SaslMechanismSelector.fromString\(string\)](#) メソッドからの構文を使用して SASL メカニズムを指定します。
この要素には以下の属性があります。

属性名	属性の説明
selector	SASL メカニズムセレクター。

sasl-mechanism-selector に必要な文法に関する詳細は、JBoss EAP 『[How to Configure Server Security](#)』の「**sasl-mechanism-selector Grammar**」を参照してください。

set-mechanism-properties

この要素には、認証メカニズムに渡される1つまたは複数の **property** 要素を含めることができます。

property

この要素は、認証メカニズムに渡されるプロパティを定義します。
この要素は以下の属性を持ちます。

属性名	属性の説明
key	プロパティ名。
value	プロパティの値。

credentials

この要素は、認証中に使用できる1つまたは複数のクレデンシャルを定義します。クレデンシャルストアの実装に依存する以下の子要素を1つ以上含めることができます。

- [key-store-reference](#)
- [credential-store-reference](#)
- [clear-password](#)
- [key-pair](#)
- [certificate](#)
- [public-key-pem](#)
- [bearer-token](#)
- [oauth2-bearer-token](#).

これらは、[protection-parameter-credentials](#) 要素に含まれる子要素と同じです。詳細と設定例は、[protection-parameter-credentials](#) 要素の説明を参照してください。

set-authorization-name

この要素は、承認に使用する名前が認証 ID とは異なる場合に、その名前を指定します。この要素は以下の属性を持ちます。

属性名	属性の説明
name	承認に使用する名前。

use-provider-sasl-factory

継承または設定に定義され、利用可能な SASL クライアントファクトリーの検索に使用される `java.security.Provider` インスタンスを指定します。この要素は [use-service-loader-sasl-factory](#) 要素とは使用しないでください。

use-service-loader-sasl-factory

この要素は、サービスローダー検出メカニズムを使用して SASL クライアントファクトリーの検出に使用されるモジュールを指定します。指定されたモジュールがない場合は、設定をロードしたクラスローダーが使用されます。この要素は [use-provider-sasl-factory](#) 要素とは使用しないでください。

この要素には以下の属性があります。

属性名	属性の説明
module-name	モジュール名

net-authenticator

この要素に含まれる設定はありません。この要素が存在する場合、`org.wildfly.security.auth.util.ElytronAuthenticator` が `java.net.Authenticator.setDefault(Authenticator)` で登録されます。これにより、認証を必要とする HTTP 呼び出しに JDK API が使用された場合に Elytron 認証クライアント設定が認証に使用されません。



注記

JDK は JBM 全体で最初の呼び出し時に認証をキャッシュするため、同じ URI への複数の呼び出しに異なるクレデンシャルを必要としないスタンドアロンプロセスのみに使用することが推奨されます。

ssl-context-rules

これは任意の要素で、SSL コンテキストルールを定義します。`ssl-context` が必要な場合、アクセスされたリソースの URI と、任意の抽象型および抽象型オーソリティーは、設定に定義されたルールと照合され、使用する `ssl-context` を特定します。

この要素には、1つまたは複数の子 `rule` 要素を含めることができます。

例: ssl-context-rules 設定

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <ssl-context-rules>
      <rule use-ssl-context="...">
        ...
      </rule>
    </ssl-context-rules>
    ...
  </authentication-client>
</configuration>
```

rule

この要素は、SSL コンテキスト定義で照合するルールを定義します。この要素には以下の属性があります。

属性名	属性の説明
use-ssl-context	ルールに一致したときに選択される SSL コンテキスト定義。

SSL コンテキストルールの照合は、認証ルールの照合とは独立しています。SSL コンテキストルールは SSL コンテキスト、認証ルールは認証設定を参照すること以外は、SSL コンテキストルールの構造は、認証設定ルールの構造と同じです。

これには、以下の子要素を含めることができます。

- `match-no-user`
- `match-user`
- `match-protocol`

- [match-host](#)
- [match-path](#)
- [match-port](#)
- [match-urn](#)
- [match-domain-name](#)
- [match-abstract-type](#)

例: SSL コンテキストの rule 設定

```
<rule use-ssl-context="...">
  <!-- At most one of the following two can be defined. -->
  <match-no-user />
  <match-user name="..." />
  <!-- Each of the following can be defined at most once. -->
  <match-protocol name="..." />
  <match-host name="..." />
  <match-path name="..." />
  <match-port number="..." />
  <match-urn name="..." />
  <match-domain name="..." />
  <match-abstract-type name="..." authority="..." />
</rule>
```

ssl-contexts

これは任意の要素で、SSL コンテキストルールによって選択される SSL コンテキスト定義を定義します。

例: ssl-contexts 設定

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <ssl-contexts>
      <default-ssl-context name="..." />
      <ssl-context name="...">
        <key-store-ssl-certificate>...</key-store-ssl-certificate>
        <trust-store key-store-name="..." />
        <cipher-suite selector="..." />
        <protocol names="... ..." />
        <provider-name name="..." />
        <providers>...</providers>
        <certificate-revocation-list path="..." maximum-cert-path="..." />
      </ssl-context>
    </ssl-contexts>
  </authentication-client>
</configuration>
```

default-ssl-context

この要素は、`javax.net.ssl.SSLContext.getDefault()` によって返される `SSLContext` を取り、`ssl-context-rules` から参照されるように名前を割り当てます。この要素は繰り返すことができるため、異なる名前を使用してデフォルトの SSL コンテキストを参照することができます。

ssl-context

この要素は、接続に使用する SSL コンテキストを定義します。任意で以下の各子要素を1つずつ含めることができます。

- [key-store-ssl-certificate](#)
- [trust-store](#)
- [cipher-suite](#)
- [protocol](#)
- [provider-name](#)
- [providers](#)
- [certificate-revocation-list](#)

key-store-ssl-certificate

この要素は、キーのキーストア内のエントリーへの参照と、この SSL コンテキストに使用する証明書を定義します。

この要素は以下の属性を持ちます。

属性名	属性の説明
key-store-name	キーストア名。必須の属性です。
alias	参照されたキーストアからロードするエントリーのエイリアス。単一のエントリーのみが含まれるキーストアのみ省略できます。

以下の子要素を含めることができます。

- [key-store-clear-password](#)
- [credential-store-reference](#)
- [key-store-credential](#)

この構造は、[key-store-credential](#) 設定で使用される構造とほぼ同じですが、この構造ではキーと証明書のエントリーを取得します。しかし、入れ子の [key-store-clear-password](#) および [key-store-credential](#) 要素は、エントリーをアンロックするために保護パラメーターを提供します。

例: key-store-ssl-certificate 設定

```
<key-store-ssl-certificate key-store-name="..." alias="...">
  <key-store-clear-password password="..." />
  <key-store-credential>...</key-store-credential>
</key-store-ssl-certificate>
```

trust-store

この要素は、**TrustManager** の初期化に使用されるキーストアへの参照です。
この要素には以下の属性があります。

属性名	属性の説明
key-store-name	キーストア名。必須の属性です。

cipher-suite

この要素は、有効化された暗号スイートにフィルターを設定します。
この要素には以下の属性があります。

属性名	属性の説明
selector	暗号スイートをフィルターするセレクター。セレクターは、 org.wildfly.security.ssl.CipherSuiteSelector.fromString(selector) メソッドによって作成される OpenSSL スタイルの暗号リスト文字列の形式を使用します。

例: デフォルトのフィルターを使用した cipher-suite 設定

```
<cipher-suite selector="DEFAULT" />
```

protocol

この要素は、サポートされるプロトコルのスペース区切りリストを定義します。使用できるプロトコルのリストは、JBoss EAP 『[How to Configure Server Security](#)』の「**client-ssl-context Attributes**」の表を参照してください。Red Hat は **TLSv1.2** の使用を推奨します。

provider-name

使用できるプロバイダーの特定後、この要素で定義された名前を持つプロバイダーのみが使用されます。

certificate-revocation-list

この要素は、証明書失効リストへのパスと、証明書パスに存在可能な自己発行でない中間証明書の最大数の両方を定義します。この要素が存在する場合、ピア証明書を証明書失効リストと照合することができます。

この要素は以下の属性を持ちます。

属性名	属性の説明
path	証明書リストへのパス。この属性は任意です。
maximum-cert-path	証明書パスに存在可能な自己発行でない中間証明書の最大数。この属性は任意です。

providers

この要素は、必要時に `java.security.Provider` インスタンスを探す方法を定義します。これには、以下の子要素を含めることができます。

- `global`
- `use-service-loader`

`authentication-client` の設定セクションはお互いに独立しているため、この要素は以下の場所に設定できます。

例: providers 設定の場所

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <providers />
    ...
    <credential-stores>
      <credential-store name="...">
        ...
        <providers />
      </credential-store>
    </credential-stores>
    ...
    <authentication-configurations>
      <authentication-configuration name="...">
        ...
        <providers />
      </authentication-configuration>
    </authentication-configurations>
    ...
    <ssl-contexts>
      <ssl-context name="...">
        ...
        <providers />
      </ssl-context>
    </ssl-contexts>
  </authentication-client>
</configuration>
```

`providers` 設定は、オーバーライドされない限りは定義される要素と、その要素の子要素に適用されます。子要素の `providers` の仕様は、その親要素に指定された `providers` をオーバーライドします。指定された `providers` 設定がない場合、デフォルトの動作は以下と同様になり、Elytron プロバイダーはグローバルに登録されたプロバイダーよりも優先されますが、グローバルに登録されたプロバイダーの使用も許可されます。

例: providers 設定

```
<providers>
  <use-service-loader />
  <global />
</providers>
```

global

この空の要素は、`java.security.Security.getProviders()` メソッド呼び出しによってロードされるグローバルプロバイダーの使用を指定します。

use-service-loader

この空の要素は、指定のモジュールによってロードされるプロバイダーの使用を指定します。指定されたモジュールがない場合は、認証クライアントをロードしたクラスローダーが使用されます。



重要

JBoss EAP の認証メカニズムによって現在使用されていない要素

Elytron クライアント設定の **credentials** 要素の以下の子要素は、JBoss EAP の認証メカニズムによって現在使用されていません。認証メカニズムのカスタム実装で使用することはできますが、サポートはされません。

1. **key-pair**
2. **public-key-pem**
3. **key-store-reference**
4. **certificate**

18.1.2. wildfly-config.xml ファイルを使用した EJB クライアント設定

`urn:jboss:wildfly-client-ejb:3.0` ネームスペースにある **jboss-ejb-client** 要素を使用し、**wildfly-config.xml** ファイルを使用すると、EJB クライアント接続、グローバルインターセプター、および呼び出しのタイムアウトを設定できます。ここでは、この要素を使用した EJB クライアントの設定方法について説明します。

jboss-ejb-client 要素および属性

jboss-ejb-client 要素には、任意で以下の3つのトップレベル子要素と、それらの子要素を含めることができます。

- **invocation-timeout**
- **global-interceptors**
 - **interceptor**
- **connections**
 - **connection**
 - **interceptors**
 - **interceptor**

invocation-timeout

これは任意の要素で、EJB 呼び出しのタイムアウトを指定します。この要素には以下の属性があります。

属性名	属性の説明
-----	-------

属性名	属性の説明
seconds	EJB ハンドシェイクまたはメソッド呼び出しのリクエストと応答のサイクルに対するタイムアウト値 (秒単位)。必須の属性です。 メソッドの実行時間がタイムアウトの時間よりも長くなった場合、呼び出しによって jboss-ejb-client が発生しますが、サーバー側は中断されません。

global-interceptors

これは任意の要素で、グローバル EJB クライアントインターセプターを指定します。任意の数の **interceptor** 要素を含めることができます。

interceptor

この要素は、EJB クライアント接続を指定するために使用されます。この要素は以下の属性を持ちます。

属性名	属性の説明
class	org.jboss.ejb.client.EJBClientInterceptor インターフェースを実装するクラスの名前。必須の属性です。
module	インターセプタークラスのロードに使用されるモジュールの名前。この属性は任意です。

connections

この要素は、EJB クライアント接続を指定するために使用されます。任意の数の **connection** 要素を含めることができます。

connection

この要素は、EJB クライアント接続を指定するために使用されます。任意で **interceptors** 要素を含めることができます。この要素には以下の属性があります。

属性名	属性の説明
uri	接続の宛先 URI。必須の属性です。

interceptors

この要素は、EJB クライアントインターセプターを指定するために使用されます。任意の数の **interceptor** 要素を含めることができます。

wildfly-config.xml ファイルの EJB クライアント設定例

以下は、**wildfly-config.xml** ファイルの **jboss-ejb-client** 要素を使用して、EJB クライアント、グローバルインターセプター、および呼び出しタイムアウトを設定する例になります。

```
<configuration>
...
  <jboss-ejb-client xmlns="urn:jboss:wildfly-client-ejb:3.0">
```

```

<invocation-timeout seconds="10"/>
<connections>
  <connection uri="remote+http://10.20.30.40:8080"/>
</connections>
<global-interceptors>
  <interceptor class="org.jboss.example.ExampleInterceptor"/>
</global-interceptors>
</jboss-ejb-client>
...
</configuration>

```

18.1.3. wildfly-config.xml ファイルを使用した HTTP クライアント設定

以下は、**wildfly-config.xml** ファイルを使用して HTTP クライアントを設定する方法の例になります。

```

<configuration>
...
  <http-client xmlns="urn:wildfly-http-client:1.0">
    <defaults>
      <eagerly-acquire-session value="true" />
      <buffer-pool buffer-size="2000" max-size="10" direct="true" thread-local-size="1" />
    </defaults>
  </http-client>
...
</configuration>

```

重要

wildfly-config.xml ファイルを使用した HTTP クライアント設定は、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。

18.1.4. wildfly-config.xml ファイルを使用したリモートクライアント設定

urn:jboss-remoting:5.0 ネームスペースにある **endpoint** 要素を使用すると、**wildfly-config.xml** ファイルを使用してリモートクライアントを設定できます。ここでは、この要素を使用したリモートクライアントの設定方法について説明します。

endpoint 要素および属性

endpoint 要素には、任意で以下の 2 つのトップレベル子要素と、それらの子要素を含めることができます。

- **providers**
 - **provider**
- **connections**
 - **connection**

さらに、以下の属性があります。

属性名	属性の説明
name	エンドポイント名。この属性は任意です。指定がない場合、可能であればエンドポイント名はシステムのホスト名から派生します。

providers

これは任意の要素で、リモートエンドポイントのトランスポートプロバイダーを指定します。任意の数の **provider** を含めることができます。

provider

この要素は、リモートトランスポートプロバイダーを定義します。この要素は以下の属性を持ちます。

属性名	属性の説明
scheme	このプロバイダーに対応するプライマリー URI スキーム。必須の属性です。
aliases	このプロバイダーに対しても認識される他の URI スキーム名のスペース区切りリスト。この属性は任意です。
module	プロバイダー実装が含まれるモジュールの名前。この属性は任意です。指定がない場合は、JBoss Remoting をロードするクラスローダーがプロバイダークラスを検索します。
class	トランスポートプロバイダーを実装するクラスの名前。この属性は任意です。指定がない場合は、プロバイダークラスの検索に java.util.ServiceLoader ファシリティーが使用されます。

connections

これは任意の要素で、リモートエンドポイントの接続を指定します。任意の数の **connection** 要素を含めることができます。

connection

この要素は、リモートエンドポイントの接続を定義します。この要素は以下の属性を持ちます。

属性名	属性の説明
destination	エンドポイントの宛先 URI。必須の属性です。
read-timeout	対応するソケットの読み取り操作に対するタイムアウト値 (秒単位)。この属性は任意ですが、 heartbeat-interval が定義されている場合のみ指定してください。
write-timeout	書き込み操作に対するタイムアウト値 (秒単位)。この属性は任意ですが、 heartbeat-interval が定義されている場合のみ指定してください。

属性名	属性の説明
ip-traffic-class	この接続のトラフィックに使用する数値の IP トラフィッククラスを定義します。この属性は任意です。
tcp-keepalive	TCP キープアライブを使用するかどうかを決定するブール値設定。この属性は任意です。
heartbeat-interval	接続ハートビートのチェック時に使用する間隔 (ミリ秒単位)。この属性は任意です。

wildfly-config.xml ファイルのリモートクライアント設定例

以下は、**wildfly-config.xml** ファイルを使用してリモートクライアントを設定する例になります。

```
<configuration>
...
<endpoint xmlns="urn:jboss-remoting:5.0">
  <connections>
    <connection destination="remote+http://10.20.30.40:8080" read-timeout="50" write-timeout="50"
heartbeat-interval="10000"/>
  </connections>
</endpoint>
...
</configuration>
```

18.1.5. wildfly-config.xml ファイルを使用したデフォルトの XNIO ワーカー設定

urn:xnio:3.5 ネームスペースにある **worker** 要素を使用すると、**wildfly-config.xml** ファイルを使用して XNIO ワーカーを設定できます。ここでは、この要素を使用して XNIO ワーカークライアントを設定する方法を説明します。

worker 要素および属性

worker 要素には、任意で以下のトップレベル子要素とそれらの子要素を含むことが可能です。

- **daemon-threads**
- **worker-name**
- **pool-size**
- **task-keepalive**
- **io-threads**
- **stack-size**
- **outbound-bind-addresses**
 - **bind-address**

daemon-threads

これは任意の要素で、ワーカーおよびタスクスレッドがデーモンスレッドであるべきかどうかを指定します。この要素の内容はありません。この要素には以下の属性があります。

属性名	属性の説明
value	ワーカーおよびタスクスレッドがデーモンスレッドであるべきかどうかを指定するブール値。 true の値はワーカーおよびタスクスレッドがデーモンスレッドであるべきであることを示します。 false の場合は、デーモンスレッドであるべきでないことを示します。必須の属性です。 この要素の指定がない場合、 true の値が想定されます。

worker-name

この要素はワーカーの名前を定義します。ワーカー名はスレッドダンプおよび JMX に表示されます。この要素の内容はありません。この要素には以下の属性があります。

属性名	属性の説明
value	ワーカーの名前。必須の属性です。

pool-size

これは任意の要素で、ワーカーのタスクスレッドプールの最大サイズを定義します。この要素の内容はありません。この要素には以下の属性があります。

属性名	属性の説明
max-threads	作成するスレッドの最大数を指定する正の整数。これは必須の属性です。

task-keepalive

これはオプションの要素で、タスクスレッドが期限切れになる前にキープアライブの時間を指定します。この要素には以下の属性があります。

属性名	属性の説明
value	アイドル状態のスレッドを保持する最小期間を秒数で指定する正の整数。これは必須の属性です。

io-threads

これは任意の要素で、維持する I/O セレクタースレッドの数を決定します。通常この数は、利用可能なコアの数の倍数で、小さな定数になります。この要素には以下の属性があります。

属性名	属性の説明
value	I/O スレッドの数を指定する正の整数。これは必須の属性です。

stack-size

これは任意の要素で、ワーカースレッドの最低スレッドスタックサイズを指定します。この要素は、密度が非常に高い特殊な状態のみで定義する必要があります。この要素には以下の属性があります。

属性名	属性の説明
value	要求されたスタックサイズをバイト単位で指定する正の整数。これは必須の属性です。

outbound-bind-addresses

これは任意の要素で、アウトバウンド接続に使用するバインドアドレスを指定します。各バインドアドレスマッピングは、宛先 IP アドレスブロックと、そのブロック内の宛先への接続に使用するバインドアドレスおよびオプションのポート番号で構成されます。任意の数の **bind-address** 要素を含めることができます。

bind-address

これは任意の要素で、個別のバインドアドレスマッピングを定義します。この要素は以下の属性を持ちます。

属性名	属性の説明
match	照合する CIDR 表記の IP アドレスブロック。
bind-address	アドレスブロックが一致した場合にバインドする IP アドレス。これは必須の属性です。
bind-port	アドレスブロックが一致した場合にバインドするポート番号。この値のデフォルトは 0 で、任意のポートにバインドします。この属性は任意です。

wildfly-config.xml ファイルの XNIO ワーカー設定例

以下は、**wildfly-config.xml** ファイルを使用してデフォルトの XNIO ワーカーを設定する方法の例になります。

```
<configuration>
...
<worker xmlns="urn:xnio:3.5">
  <io-threads value="10"/>
  <task-keepalive value="100"/>
</worker>
...
</configuration>
```

第19章 ECLIPSE MICROPROFILE

19.1. ECLIPSE MICROPROFILE OPENTRACING を使用したリクエストのトレース



重要

Eclipse Microprofile OpenTracing は、テクノロジープレビューとしてのみ提供されません。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。

Eclipse Microprofile OpenTracing 機能は **microprofile-opentracing-smallrye** サブシステムによって提供されます。このサブシステムには **Jaeger Java Client** がデフォルトのトレーサーとして同梱され、JAX-RS や CDI など Java EE アプリケーションで一般的に使用されるコンポーネントのインスツルメンテーションライブラリーのセットを提供します。このサブシステムに関する詳細は、『[設定ガイド](#)』の「[MicroProfile OpenTracing SmallRye サブシステムでのリクエストのトレース](#)」を参照してください。

これ以降のセクションでは、[CDI bean](#) およびその他の [JAX-RS エンドポイント](#) のトレースをカスタマイズする方法や、[カスタムトレーサー](#) の実装方法について説明します。

19.1.1. CDI bean のトレースの有効または無効化

Eclipse MicroProfile OpenTracing 仕様で定義されているように、クラスまたはメソッドレベルで [org.eclipse.microprofile.opentracing.Traced](#) アノテーションが存在すると CDI bean はトレースされます。アノテーションの値を **false** に設定するとトレースを無効にできます。同様に、そのアノテーションの **operationName** パラメーターを指定すると、カスタムの操作名を設定できます。セマンティックは [MicroProfile OpenTracing 仕様](#) によって定義されます。

以下の例は、CDI bean のトレースの設定方法を表しています。トレースはメソッドレベルで指定することに注意してください。

```
import org.eclipse.microprofile.opentracing.Traced;

@Traced
public class TracedBean {
    public void doSomething() {
    }

    @Traced(true)
    public void doSomethingTraced() {
    }

    @Traced(false)
    public void doSomethingNotTraced() {
    }
}
```

以下の例は、このトレースポイントの OpenTracing Span の操作名を指定する方法を表しています。

```
import org.eclipse.microprofile.opentracing.Traced;

@Traced(operationName = "my-custom-class-operation-name")
public class CustomOperationNameBean {

    @Traced(operationName = "my-custom-method-operation-name")
    public void doSomething() {
    }

    @Traced
    public void doSomethingElse() {
    }
}
```

19.1.2. JAX-RS エンドポイントのトレースの有効または無効化

JAX-RS エンドポイントは、サーバー設定に **microprofile-opentracing-smallrye** サブシステムが存在する場合はデフォルトでトレースされます。

JAX-RS エンドポイントのトレースを無効にするには、「**CDI bean のトレースの有効または無効化**」に記載されているとおりに、クラスまたはメソッドレベルで JAX-RS エンドポイントに `@Traced(false)` アノテーションを追加します。

19.1.3. カスタムトレーサーの実装

デフォルトの Jaeger Java Client よりも複雑なトレーサーが必要な場合は、希望の状態の **Tracer** を返す **TracerResolver** を実装して独自のトレーサーを提供することができます。この場合、デフォルトのトレーサーは使用されません。

以下の例は、**Tracer** クラスのカスタム実装を返す **TracerResolver** の新しい実装を作成する方法を表しています。

```
import io.opentracing.Tracer;
import io.opentracing.contrib.tracerresolver.TracerResolver;
import org.myproject.opentracing.CustomTracer;

public static class MyTracerResolver extends TracerResolver {
    @Override
    protected Tracer resolve() {
        return new CustomTracer();
    }
}
```

19.2. ECLIPSE MICROPROFILE HEALTH を使用したサーバー状態の監視

重要

Eclipse MicroProfile Health 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。

Eclipse Microprofile Health 機能は **microprofile-health-smallrye** サブシステムによって提供されます。このサブシステムに関する詳細は、『[設定ガイド](#)』の「[Eclipse MicroProfile Health を使用したサーバー状態の監視](#)」を参照してください。

これ以降のセクションでは、[カスタムヘルスチェックの実装方法](#)について説明します。

19.2.1. カスタムヘルスチェックの実装

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、以下のように、後続のヘルスチェッククエリーすべて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"
    }]
  }
}
```

付録A リファレンス資料

A.1. 提供される UNDERTOW ハンドラー



注記

ハンドラーの完全リストは、お使いの JBoss EAP インストールの Undertow コアと一致するバージョンの Undertow コアのソース JAR ファイルを確認してください。Undertow コアのソース JAR ファイルは [JBoss EAP Maven リポジトリ](#) からダウンロードでき、利用できるハンドラーは `/io/undertow/server/handlers/` ディレクトリーで参照できます。

現在の JBoss EAP インストールで使用されている Undertow コアのバージョンを確認するには、JBoss EAP サーバーの起動中に出力される、以下と似た **INFO** メッセージを `server.log` ファイルで検索します。

```
INFO [org.wildfly.extension.undertow] (MSC service thread 1-1) WFLYUT0003:
Undertow 1.4.18.Final-redhat-1 starting
```

AccessControlListHandler

クラス名: `io.undertow.server.handlers.AccessControlListHandler`

名前: `access-control`

リモートピアの属性に基づいて要求を受諾または拒否できるハンドラー。

表A.1 パラメーター

名前	説明
<code>acl</code>	ACL ルール。このパラメーターは必須です。
<code>attribute</code>	Exchange 属性文字列。このパラメーターは必須です。
<code>default-allow</code>	ハンドラーがデフォルトで要求を受諾または拒否するかどうかを指定するブール値。デフォルトは false です。

AccessLogHandler

クラス名: `io.undertow.server.handlers.accesslog.AccessLogHandler`

名前: `access-log`

アクセスログハンドラー。このハンドラーは、提供された書式文字列に基づいてアクセスログメッセージを生成し、提供された **AccessLogReceiver** にそれらのメッセージを渡します。

このハンドラーは、**ExchangeAttribute** メカニズムにより提供されたすべての属性をログに記録できます。

このファクトリーは、以下のパターンのトークンハンドラーを生成します。

表A.2 パターン

パターン	説明
%A	リモート IP アドレス
%A	ローカル IP アドレス
%b	送信済みバイト数 (HTTP ヘッダーまたは - を除く (バイトが送信されなかった場合))
%B	送信済みバイト数 (HTTP ヘッダーを除く)
%h	リモートホスト名
%H	要求プロトコル
%l	identd からのリモート論理ユーザー名 (常に - を返します)
%m	要求メソッド
%p	ローカルポート
%q	クエリー文字列 (? 文字を除く)
%r	要求の最初の行
%s	応答の HTTP ステータスコード
%t	Common Log Format 形式の日時
%u	認証されたリモートユーザー
%U	要求された URL パス
%v	ローカルサーバー名
%D	要求を処理するのにかかった時間 (ミリ秒単位)
%T	要求を処理するのにかかった時間 (秒単位)
%I	現在の要求スレッド名 (後でスタックトレースと比較できます)
common	%h %l %u %t "%r" %s %b

パターン	説明
combined	<code>%h %l %u %t "%r" %s %b "%{i,Referer}" "%{i,User-Agent}"</code>

クッキー、受信ヘッダー、またはセッションから情報を書き込むこともできます。

Apache 構文に基づきます。

- `%{i,xxx}` (受信ヘッダーの場合)
- `%{o,xxx}` (送信応答ヘッダーの場合)
- `%{c,xxx}` (特定のクッキーの場合)
- `%{r,xxx}` (`xxx` は `ServletRequest` の属性)
- `%{s,xxx}` (`xxx` は `HttpSession` の属性)

表A.3 パラメーター

名前	説明
format	ログメッセージを生成するために使用する形式。これはデフォルトパラメーターです。

AllowedMethodsHandler

特定の HTTP メソッドのホワイトリストに登録するハンドラー。許可されたメソッドセットの1つを持つリクエストのみが許可されます。

クラス名: `io.undertow.server.handlers.AllowedMethodsHandler`

名前: `allowed-methods`

表A.4 パラメーター

名前	説明
methods	許可されるメソッド (<code>GET</code> 、 <code>POST</code> 、 <code>PUT</code> など)。これはデフォルトパラメーターです。

BlockingHandler

ブロック要求を開始する `HttpHandler`。スレッドが現在 I/O スレッドで実行されている場合、スレッドはディスパッチされます。

クラス名: `io.undertow.server.handlers.BlockingHandler`

名前: `blocking`

このハンドラーにはパラメーターがありません。

ByteRangeHandler

範囲要求のハンドラー。これは、修正されたコンテンツの長さのリソース (たとえば、**content-length** ヘッダーが設定されたリソース) に対する範囲要求を処理できる汎用ハンドラーです。コンテンツすべてが生成され、破棄されるため、これは必ずしも範囲要求を処理する最も効率的な方法ではありません。現時点では、このハンドラーは単純な単一の範囲要求しか処理できません。複数の範囲が要求された場合は、**Range** ヘッダーが無視されます。

クラス名: **io.undertow.server.handlers.ByteRangeHandler**

名前: **byte-range**

表A.5 パラメーター

名前	説明
send-accept-ranges	承認範囲を送信するかどうかを決定するブール値。これはデフォルトパラメーターです。

CanonicalPathHandler

このハンドラーは、相対パスを正規のパスに変換します。

クラス名: **io.undertow.server.handlers.CanonicalPathHandler**

名前: **canonical-path**

このハンドラーにはパラメーターがありません。

DisableCacheHandler

ブラウザおよびプロキシによる応答キャッシュを無効にするハンドラー。

クラス名: **io.undertow.server.handlers.DisableCacheHandler**

名前: **disable-cache**

このハンドラーにはパラメーターがありません。

DisallowedMethodsHandler

特定の HTTP メソッドをブラックリストに登録するハンドラー。

クラス名: **io.undertow.server.handlers.DisallowedMethodsHandler**

名前: **disallowed-methods**

表A.6 パラメーター

名前	説明
methods	許可しないメソッド (たとえば、 GET 、 POST 、 PUT など)。これはデフォルトパラメーターです。

EncodingHandler

このハンドラーは、コンテンツのエンコーディング実装の基礎となります。このハンドラーに委譲するものとして、エンコーディングハンドラーが、指定されたサーバー側の優先度で追加されます。

正しいハンドラーを決定するために **q** 値が使用されます。**q** 値なしで要求が行われた場合、サーバーは使用するエンコーディングとして最も優先度が高いハンドラーを選択します。

一致するハンドラーがない場合は、ID エンコーディングが選択されます。**q** 値が **0** であるため、ID エンコーディングが特別に許可されない場合は、ハンドラーにより応答コード **406 (Not Acceptable)** が設定され、返されます。

クラス名: `io.undertow.server.handlers.encoding.EncodingHandler`

名前: `compress`

このハンドラーにはパラメーターがありません。

FileErrorHandler

エラーページとして使用するファイルをディスクから提供するハンドラー。このハンドラーはデフォルトで応答コードを提供しません。応答コードは設定する必要があります。

クラス名: `io.undertow.server.handlers.error.FileErrorHandler`

名前: `error-file`

表A.7 パラメーター

名前	説明
file	エラーページとして使用するファイルの場所。
response-codes	定義されたエラーページファイルにリダイレクトする応答コードのリスト。

HttpTraceHandler

HTTP トレース要求を処理するハンドラー。

クラス名: `io.undertow.server.handlers.HttpTraceHandler`

名前: `trace`

このハンドラーにはパラメーターがありません。

IPAddressAccessControlHandler

リモートピアの IP アドレスに基づいて要求を受諾または拒否できるハンドラー。

クラス名: `io.undertow.server.handlers.IPAddressAccessControlHandler`

名前: `ip-access-control`

表A.8 パラメーター

名前	説明
acl	アクセス制御リストを表す文字列。これは デフォルトパラメーター です。

名前	説明
failure-status	拒否された要求で返されるステータスコードを表す文字列。
default-allow	デフォルトで許可するかどうかを表すブール値。

JDBCLogHandler

クラス名: `io.undertow.server.handlers.JDBCLogHandler`

名前: `jdbc-access-log`

表A.9 パラメーター

名前	説明
format	JDBC ログパターンを指定します。デフォルト値は common です。また、 combined を使用して、VirtualHost、要求メソッド、参照元、およびユーザーエージェント情報をログメッセージに追加することもできます。
datasource	ログするデータソースの名前。このパラメーターは必須であり、 デフォルトパラメーター です。
tableName	テーブル名。
remoteHostField	リモートホストアドレス。
userField	ユーザー名。
timestampField	タイムスタンプ。
virtualHostField	VirtualHost。
methodField	メソッド。
queryField	クエリー。
statusField	ステータス。
bytesField	バイト数。
refererField	参照元。
userAgentField	UserAgent。

LearningPushHandler

ブラウザが要求するリソースのキャッシュを構築し、サーバープッシュを使用してリソースをプッシュ (サポートされている場合) するハンドラー。

クラス名: `io.undertow.server.handlers.LearningPushHandler`

名前: `learning-push`

表A.10 パラメーター

名前	説明
<code>max-age</code>	キャッシュエントリーの最大期間を表す整数。
<code>max-entries</code>	キャッシュエントリーの最大数を表す整数。

LocalNameResolvingHandler

DNS ルックアップを実行してローカルアドレスを解決するハンドラー。フロントエンドサーバーが **X-forwarded-host** ヘッダーを送信した場合、または AJP が使用中の場合は、未解決のローカルアドレスが作成されることがあります。

クラス名: `io.undertow.server.handlers.LocalNameResolvingHandler`

名前: `resolve-local-name`

このハンドラーにはパラメーターがありません。

PathSeparatorHandler

URL のスラッシュでない区切り文字をスラッシュに変換するハンドラー。一般的に、Windows システムではバックスラッシュはスラッシュに変換されます。

クラス名: `io.undertow.server.handlers.PathSeparatorHandler`

名前: `path-separator`

このハンドラーにはパラメーターがありません。

PeerNameResolvingHandler

リバース DNS ルックアップを実行してピアアドレスを解決するハンドラー。

クラス名: `io.undertow.server.handlers.PeerNameResolvingHandler`

名前: `resolve-peer-name`

このハンドラーにはパラメーターがありません。

ProxyPeerAddressHandler

X-Forwarded-For ヘッダーの値にピアアドレスを設定するハンドラー。これは、このヘッダーを常に設定するプロキシの背後でのみ使用してください。そのように使用しないと、攻撃者がピアアドレスを偽造することがあります。

クラス名: `io.undertow.server.handlers.ProxyPeerAddressHandler`

名前: `proxy-peer-address`

このハンドラーにはパラメーターがありません。

RedirectHandler

302 リダイレクトを使用して、指定された場所にリダイレクトするリダイレクトハンドラー。場所は `exchange` 属性文字列として指定されます。

クラス名 `io.undertow.server.handlers.RedirectHandler`

名前: `redirect`

表A.11 パラメーター

名前	説明
value	リダイレクトの宛先。これはデフォルトパラメーターです。

RequestBufferingHandler

すべての要求データをバッファするハンドラー。

クラス名: `io.undertow.server.handlers.RequestBufferingHandler`

名前: `buffer-request`

表A.12 パラメーター

名前	説明
buffers	バッファの最大数を定義する整数。これはデフォルトパラメーターです。

RequestDumpingHandler

`exchange` をログにダンプするハンドラー。

クラス名: `io.undertow.server.handlers.RequestDumpingHandler`

名前: `dump-request`

このハンドラーにはパラメーターがありません。

RequestLimitingHandler

同時リクエストの最大数を制限するハンドラー。この制限を超えたリクエストは、前のリクエストが完了するまでブロックされます。

クラス名: `io.undertow.server.handlers.RequestLimitingHandler`

名前: `request-limit`

表A.13 パラメーター

名前	説明
requests	同時リクエストの最大数を表す整数。これはデフォルトパラメーターであり、必須です。

ResourceHandler

リソースを提供するハンドラー。

クラス名: `io.undertow.server.handlers.resource.ResourceHandler`

名前: `resource`

表A.14 パラメーター

名前	説明
location	リソースの場所。これはデフォルトパラメーターであり、必須です。
allow-listing	ディレクトリーのリストを許可するかどうかを決定するブール値。

ResponseRateLimitingHandler

設定された数のバイト/時間にダウンロードレートを制限するハンドラー。

クラス名: `io.undertow.server.handlers.ResponseRateLimitingHandler`

名前: `response-rate-limit`

表A.15 パラメーター

名前	説明
bytes	ダウンロードレートを制限するバイトの数。このパラメーターは必須です。
time	ダウンロードレートを制限する時間 (秒単位)。このパラメーターは必須です。

SetHeaderHandler

修正された応答ヘッダーを設定するハンドラー。

クラス名: `io.undertow.server.handlers.SetHeaderHandler`

名前: `header`

表A.16 パラメーター

名前	説明
header	ヘッダー属性の名前。このパラメーターは必須です。
value	ヘッダー属性の値。このパラメーターは必須です。

SSLHeaderHandler

以下のヘッダーに基づいて接続の SSL 情報を設定するハンドラー。

- `SSL_CLIENT_CERT`
- `SSL_CIPHER`
- `SSL_SESSION_ID`

このハンドラーがチェーンに存在する場合は、これらのヘッダーが存在しなくても SSL セッション情報が常に上書きされます。

このハンドラーは、リクエストごとに常にヘッダーを指定するよう設定されている、あるいは SSL 情報が存在しない場合に、これらの名前を持つ既存のヘッダーを削除するように設定されている、リバースプロキシの背後にあるサーバー上でのみ使用する**必要があります**。このように使用しないと、悪意のあるクライアントが SSL 接続を偽装できる可能性があります。

クラス名: `io.undertow.server.handlers.SSLHeaderHandler`

名前: `ssl-headers`

このハンドラーにはパラメーターがありません。

StuckThreadDetectionHandler

このハンドラーは処理に時間がかかるリクエストを検出します (処理中のスレッドが停止していることを示すことがあります)。

クラス名: `io.undertow.server.handlers.StuckThreadDetectionHandler`

名前: `stuck-thread-detector`

表A.17 パラメーター

名前	説明
threshold	リクエストを処理する時間のしきい値を決定する整数値 (秒単位)。デフォルト値は 600 (10 分) です。これは デフォルトパラメーター です。

URLDecodingHandler

指定された文字セットに URL およびクエリーパラメーターをデコードするハンドラー。このハンドラーを使用している場合は、`UndertowOptions.DECODE_URL` パラメーターを **false** に設定する必要があります。

これはパーサーの組み込み UTF-8 デコーダーを使用する場合よりも効率的ではありません。UTF-8 以外の文字セットにデコードする必要がない限り、パーサーのデコードを使用してください。

クラス名: `io.undertow.server.handlers.URLDecodingHandler`

名前: `url-decoding`

表A.18 パラメーター

名前	説明
----	----

名前	説明
charset	デコードする文字セット。これはデフォルトパラメーターであり、必須です。

A.2. 永続ユニットプロパティ

永続ユニット定義は、**persistence.xml** ファイルから設定できる以下のプロパティをサポートします。

プロパティ	説明
jboss.as.jpa.providerModule	永続プロバイダーモジュールの名前。デフォルトは org.hibernate 。永続プロバイダーがアプリケーションとパッケージ化されている場合はアプリケーション名になります。
jboss.as.jpa.adapterModule	JBoss EAP が永続プロバイダーと動作するようにする統合クラスの名前。
jboss.as.jpa.adapterClass	統合アダプターのクラス名。
jboss.as.jpa.managed	コンテナ管理の JPA による永続ユニットへのアクセスを無効にするには false を設定します。デフォルトは true です。
jboss.as.jpa.classtransformer	永続ユニットのクラストランスフォーマーを無効にするには false を設定します。デフォルトは true で、クラスのトランスフォームを許可します。 クラスのトランスフォームを有効にするには、Hibernate の永続ユニットプロパティ hibernate.ejb.use_class_enhancer を true にする必要もあります。
jboss.as.jpa.scopedname	使用されるアプリケーションスコープの修飾永続ユニット名を指定します。デフォルトでは、アプリケーション名と永続ユニット名の組み合わせに設定されます。 hibernate.cache.region_prefix のデフォルトは、 jboss.as.jpa.scopedname の設定先になります。 jboss.as.jpa.scopedname の値は、同じアプリケーションサーバーインスタンスにデプロイされた別のアプリケーションによって使用されていない値に設定するようにしてください。
jboss.as.jpa.deferdetach	非 JTA トランザクションスレッドで使用されるトランザクションスコープの永続コンテキストが各 EntityManager 呼び出しの後にロードされたエントリをデタッチするかどうか、または永続コンテキストが閉じられるまで待機するかどうかを制御します。デフォルト値は false です。 true に設定すると、永続コンテキストが閉じられるまでデタッチが延期されます。

プロパティ	説明
wildfly.jpa.default-unit	アプリケーションのデフォルトの永続ユニットを選択する場合は true に設定します。これは、 persistence.xml ファイルに複数の永続ユニットが指定され、 unitName を指定せずに永続コンテキストをインジェクトする場合に便利です。
wildfly.jpa.twophasebootstrap	永続プロバイダーは、JPA の CDI との統合を改良する、2 フェーズ永続ユニットブートストラップを許可します。 wildfly.jpa.twophasebootstrap の値を false に設定すると、その値が含まれる永続ユニットの 2 フェーズブートストラップが無効になります。
wildfly.jpa.allowdefaultdatasource	永続ユニットがデフォルトのデータソースを使用しないようにするには、 false を設定します。デフォルト値は true です。これは、データソースを指定しない永続ユニットにのみ重要です。
wildfly.jpa.hibernate.search.module	クラスパスに含まれる Hibernate Search のバージョンを制御します。デフォルトは auto です。その他に有効な値は none と、代替バージョンを使用するためのフルモジュールの識別子です。

A.3. ポリシープロバイダープロパティ

表A.19 policy-provider 属性

プロパティ	説明
custom-policy	カスタムのポリシープロバイダー定義。
jacc-policy	JACC および関連するサービスをセットアップするポリシープロバイダー定義。

表A.20 custom-policy 属性

プロパティ	説明
class-name	ポリシープロバイダーを参照する java.security.Policy 実装の名前。
module	プロバイダーのロード元となるモジュールの名前。

表A.21 jacc-policy 属性

プロパティ	説明
policy	ポリシープロバイダーを参照する java.security.Policy 実装の名前。
configuration-factory	ポリシー設定ファクトリープロバイダーを参照する javax.security.jacc.PolicyConfigurationFactory 実装の名前。

プロパティ	説明
module	プロバイダーのロード元となるモジュールの名前。

A.4. JAVA EE 8 のプロファイルおよびテクノロジーリファレンス

以下の表では Java EE 8 のテクノロジーと JSR をカテゴリごとに表し、Web Profile または Full Platform プロファイルに含まれているかどうかを記載しています。

- [Java EE 8 の Web アプリケーションテクノロジー](#)
- [Java EE 8 のエンタープライズアプリケーションテクノロジー](#)
- [Java EE 8 の Web サービステクノロジー](#)
- [Java EE 8 の管理およびセキュリティーテクノロジー](#)
- [Java EE 8 に関連する Java SE 仕様](#)

Java EE 8 API の完全リストは「[Java™ EE 8 Technologies](#)」を参照してください

表A.22 Java EE 8 の Web アプリケーションテクノロジー

テクノロジー	JSR	Web Profile	Full Platform
Java API for WebSocket 1.1	JSR 356	✓	✓
Java API for JSON Binding 1.0	JSR 367	✓	✓
Java API for JSON Processing 1.1	JSR 374	✓	✓
Java Servlet 4.0	JSR 369	✓	✓
JavaServer Faces 2.3	JSR 372	✓	✓
Expression Language 3.0	JSR 341	✓	✓
JavaServer Pages 2.3	JSR 245	✓	✓
Standard Tag Library for JavaServer Pages (JSTL) 1.2 ¹	JSR 52	✓	✓

¹ 追加の Standard Tag Library (JSTL) 情報:

注記

JBoss EAP には既知のセキュリティーリスクが存在します。Java Standard Tag Library (JSTL) が信頼できない XML ドキュメントにおける外部エンティティ参照の処理を許可するため、ホストシステム上のリソースへアクセスし、任意コードが実行される可能性があります。

このリスクを回避するには、通常空の文字列を値として適切に設定されたシステムプロパティ **org.apache.taglibs.standard.xml.accessExternalEntity** を使用して JBoss EAP サーバーを実行する必要があります。これを行う方法は 2 つあります。

- システムプロパティを設定してサーバーを再起動する。

```
org.apache.taglibs.standard.xml.accessExternalEntity
```

- Dorg.apache.taglibs.standard.xml.accessExternalEntity=""** を引数として **standalone.sh** または **domain.sh** スクリプトに渡す。

表A.23 Java EE 8 のエンタープライズアプリケーションテクノロジー

テクノロジー	JSR	Web Profile	Full Platform
Batch Applications for the Java Platform 1.0	JSR 352		✓
Concurrency Utilities for Java EE 1.0	JSR 236		✓
Contexts and Dependency Injection for Java 2.0	JSR 365	✓	✓
Dependency Injection for Java 1.0	JSR 330	✓	✓
Bean Validation 2.0	JSR 380	✓	✓
Managed Beans 1.0	該当なし	✓	✓
Enterprise JavaBeans 3.2	JSR 345		✓
Interceptors 1.2	JSR 318	✓	✓
Java EE Connector Architecture 1.7	JSR 322		✓
Java Persistence 2.2	JSR 338	✓	✓
Common Annotations for the Java Platform 1.3	JSR 250		✓
Java Message Service API 2.0	JSR 343		✓
Java Transaction API (JTA) 1.2	JSR 907	✓	✓
JavaMail 1.6	JSR 919		✓

表A.24 Java EE 8 の Web サービステクノロジー

テクノロジー	JSR	Web Profile	Full Platform
Java API for RESTful Web Services (JAX-RS) 2.1	JSR 370		✓
Implementing Enterprise Web Services 1.3	JSR 109		✓
Web Services Metadata for the Java Platform 2.1	JSR 181		✓
Java API for XML-Based RPC (JAX-RPC) 1.1 (任意)	JSR 101		
Java API for XML Registries (JAXR) 1.0 (任意)	JSR 93		

表A.25 Java EE 8 の管理およびセキュリティーテクノロジー

テクノロジー	JSR	Web Profile	Full Platform
Java EE Security API 1.0	JSR 375	✓	✓
Java Authentication Service Provider Interface for Containers 1.1	JSR 196	✓	✓
Java Authorization Contract for Containers 1.5	JSR 115		✓
Java EE Application Deployment 1.2 (任意)	JSR 88		✓
J2EE Management 1.1	JSR 77		✓
Debugging Support for Other Languages 1.0	JSR 45		✓

表A.26 Java EE 8 に関連する Java SE 仕様

テクノロジー	JSR	Web Profile	Full Platform
Java Management Extensions (JMX) 2.0	JSR 3		✓
SOAP with Attachments API for Java (SAAJ) Specification 1.3	JSR 67		✓
Streaming API for XML (StAX) 1.0	JSR 173		✓
Java API for XML Processing (JAXP) 1.6	JSR 206		✓
Java Database Connectivity 4.0	JSR 221		✓
Java Architecture for XML Binding (JAXB) 2.2	JSR 222		✓

テクノロジー	JSR	Web Profile	Full Platform
Java API for XML-Based Web Services (JAX-WS) 2.2	JSR 224		✓
JavaBeans Activation Framework (JAF) 1.1	JSR 925		✓

Revised on 2019-11-27 12:59:48 CET