# Red Hat JBoss Enterprise Application Platform 7.3

## EJB アプリケーションの開発

Red Hat JBoss Enterprise Application Platform 用の Enterprise JavaBeans (EJB) アプリケーションの開発およびデプロイを行う開発者および管理者の手順と情報。

Last Updated: 2023-02-28

# Red Hat JBoss Enterprise Application Platform 7.3 EJB アプリケーションの開発

Red Hat JBoss Enterprise Application Platform 用の Enterprise JavaBeans (EJB) アプリケーションの開発およびデプロイを行う開発者および管理者の手順と情報。

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

## 法律上の通知



## 概要

本書は、Red Hat JBoss Enterprise Application Platform で EJB アプリケーションの開発とデプロイを行う開発者と管理者に情報を提供します。

# 目次

# 第1章 はじめに

## 1.1. EJB の概要

EJB 3.2 は、Enterprise Bean と呼ばれるサーバー側のコンポーネントを使用して、分散型のトランザクション型のセキュアな移植可能な Java EE アプリケーションを開発するための API です。エンタープライズ Bean は、再利用を促すような分離方法でアプリケーションのビジネスロジックを実装します。EJB は、Java EE 仕様の JSR 345 に記載されています。仕様の Jakarta は Jakarta Enterprise Beans 3.2 です。

EJB 3.2 は full と lite の 2 つのプロファイルを提供します。JBoss EAP 7 は、EJB 3.2 仕様を使用して構築されたアプリケーションの完全なプロファイルを実装します。

## 1.2. EJB 3.2 機能セット

以下の EJB 3.2 機能は JBoss EAP 7 でサポートされます。

- セッション Bean
- メッセージ駆動 Bean
- EJB API グループ
- インターフェイスなしビュー
- ローカルインターフェイス
- リモートインターフェイス
- AutoClosable インターフェイス
- タイマーサービス
- 非同期呼び出し
- インターセプター
- RMI/IIOP の相互運用性
- トランザクションサポート
- セキュリティー
- 組み込み API

以下の機能は JBoss EAP 7 ではサポートされなくなりました。

- EJB 2.1 エンティティー Bean クライアントビュー
- Bean 管理の永続性のあるエンティティー Bean
- コンテナー管理の永続性のあるエンティティー Bean
- EJB クエリー言語 (EJB QL)
- JAX-RPC ベースの Web サービス: エンドポイントおよびクライアントビュー

## 1.3. エンタープライズ BEAN

エンタープライズ Bean は Java クラスとして記述され、適切な EJB アノテーションが付けられます。Java EE アプリケーションまたは Jakarta EE アプリケーションの一部としてデプロイすることも、独自のアーカイブ (JAR ファイル) でアプリケーションサーバーにデプロイできます。アプリケーションサーバーは、各エンタープライズ bean のライフサイクルを管理し、セキュリティー、トランザクション、同時実行管理などのサービスを提供します。

また、エンタープライズ Bean は任意の数のビジネスインターフェイスを定義できます。ビジネスインターフェイスは、クライアントで利用できる Bean のメソッドの制御を強化し、リモート JVM で実行しているクライアントへのアクセスも許可します。

エンタープライズ Bean には、セッション Bean、メッセージ駆動 Beanエンティティー Bean の 3 つの Bean があります。

**注記**

JBoss EAP はエンティティー Bean をサポートしません。

### 1.3.1. エンタープライズ Bean の作成

エンタープライズ Bean は、Java アーカイブ (JAR) ファイルにパッケージ化され、デプロイされます。エンタープライズ Bean JAR ファイルをアプリケーションサーバーにデプロイしたり、エンタープライズアーカイブ (EAR) ファイルに組み込み、そのアプリケーションでデプロイしたりできます。また、Web アプリケーションとともに Web アーカイブ (WAR) ファイルにエンタープライズ Bean をデプロイすることもできます。

## 1.4. エンタープライズ BEAN のビジネスインターフェイス

EJB ビジネスインターフェイスは、Bean 開発者が作成する Java インターフェイスで、クライアントで利用可能なセッション Bean のパブリックメソッドの宣言を提供する 。セッション Bean は任意の数のインターフェイスを実装できます。なし ( **no-interface** bean) も含まれます。

ユーザーインターフェイスは、ローカルインターフェイスまたはリモートインターフェイスとして宣言できますが、両方に宣言することはできません。

**EJB ローカルビジネスインターフェイス**
EJB ローカルビジネスインターフェイスは、Bean とクライアントが同じ JVM にある場合に使用できるメソッドを宣言します。セッション Bean がローカルビジネスインターフェイスを実装すると、そのインターフェイスで宣言されたメソッドのみがクライアントで利用できます。

**EJB リモートビジネスインターフェイス**
EJB リモートビジネスインターフェイスは、リモートクライアントで使用できるメソッドを宣言します。リモートインターフェイスを実装するセッション Bean へのリモートアクセスは、EJB コンテナーにより自動的に提供されます。

リモートクライアントは、異なる JVM で実行されているクライアントです。また、デスクトップアプリケーションや、異なるアプリケーションサーバーにデプロイされた Web アプリケーション、サービス、およびエンタープライズ Bean を含めることができます。

ローカルクライアントは、リモートビジネスインタフェースで公開されるメソッドにアクセスできます。

**EJB No-interface Beans**

いずれのビジネスインターフェイスも実装しないセッション Bean は、インターフェイスなしの bean (no-interface bean) と呼ばれます。no-interface Bean のすべてのパブリックメソッドは、ローカルクライアントからアクセスできます。

また、ビジネスインターフェイスを実装するセッション Bean を作成して、**no-interface** ビューを公開することもできます。

## 1.5. レガシー EJB クライアントの互換性

JBoss EAP では、EJB クライアントライブラリーをプライマリー API として利用して、リモート EJB コンポーネントを呼び出します。

JBoss EAP 7.1 より、以下の 2 つの EJB クライアントが同梱されるようになりました。

- EJB クライアント: 通常の EJB クライアントは完全に後方互換性がありません。
- レガシー EJB クライアント: レガシー EJB クライアントはバイナリー後方互換性を提供します。このレガシー EJB クライアントは、JBoss EAP 7.0 の EJB クライアントを使用して最初にコンパイルされたクライアントアプリケーションで実行できます。JBoss EAP 7.0 の EJB クライアントに存在していたすべての API は、JBoss EAP 7.3 のレガシー EJB クライアントにあります。

以下の Maven 依存関係を設定に含めると、レガシー EJB クライアントの互換性を使用できます。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.eap</groupId>
      <artifactId>wildfly-ejb-client-legacy-bom</artifactId>
      <version>EAP_BOM_VERSION</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.jboss</groupId>
    <artifactId>jboss-ejb-client-legacy</artifactId>
  </dependency>
</dependencies>
```

JBoss EAP Maven リポジトリーで利用可能な **EAP_BOM_VERSION** を使用する必要があります。

# 第2章 エンタープライズ BEAN プロジェクトの作成

## 2.1. RED HAT CODEREADY STUDIO を使用した EJB アーカイブプロジェクトの作成

ここでは、Red Hat CodeReady Studio で EJB プロジェクトを作成する方法を説明します。

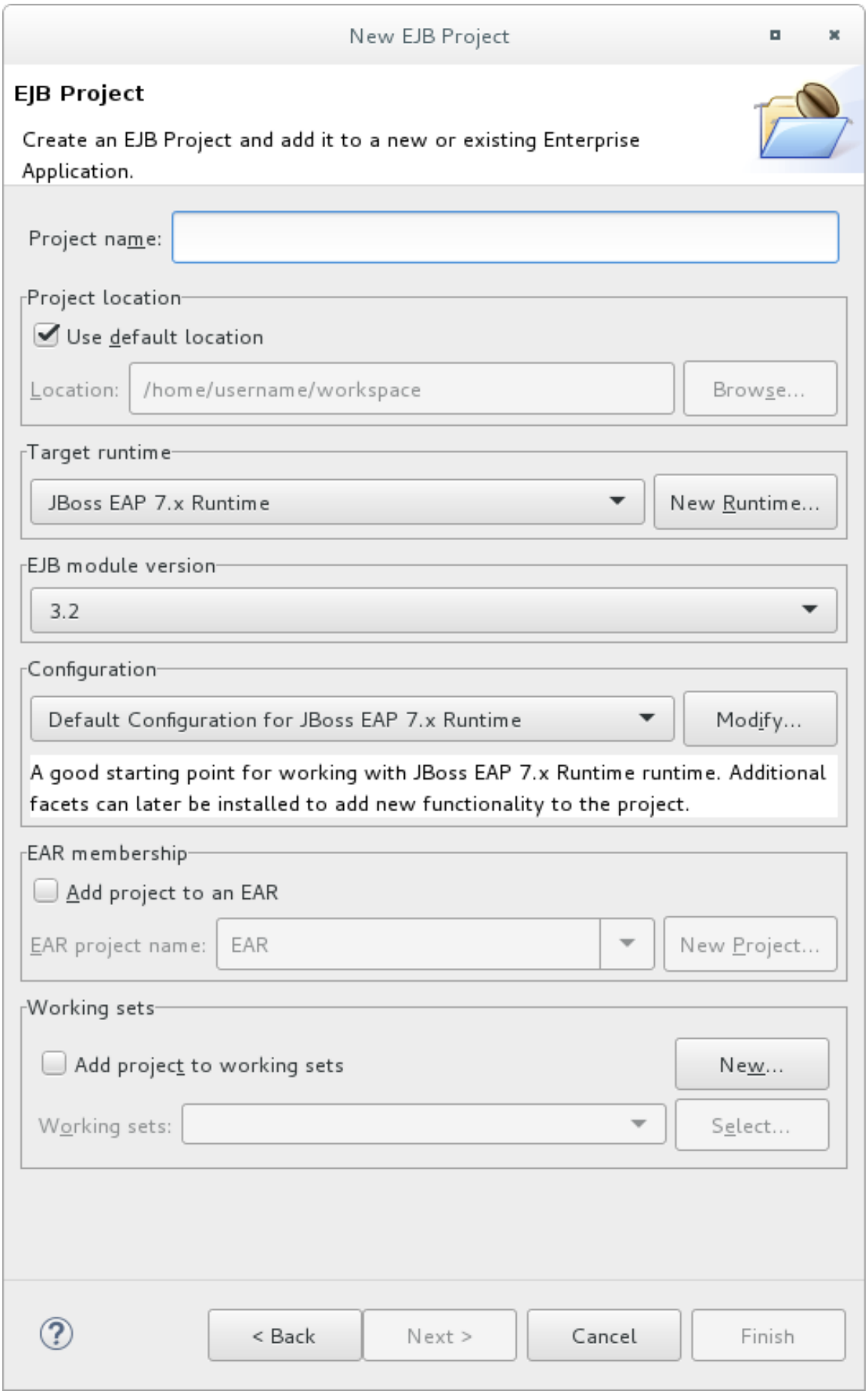
**要件**

- JBoss EAP のサーバーランタイムおよびサーバーランタイムが Red Hat CodeReady Studio で設定されています。

### Red Hat CodeReady Studio での JPA プロジェクトの作成

1. **New EJB Project** ウィザードを開きます。
   a. **File** メニューに移動して **New** を選択し、**Project** を選択します。
   b. **New Project** ウィザードが表示されたら、**EJB/EJB Project** を選択して **Next** をクリックします。

**図2.1新しい EJB プロジェクトウィザード**



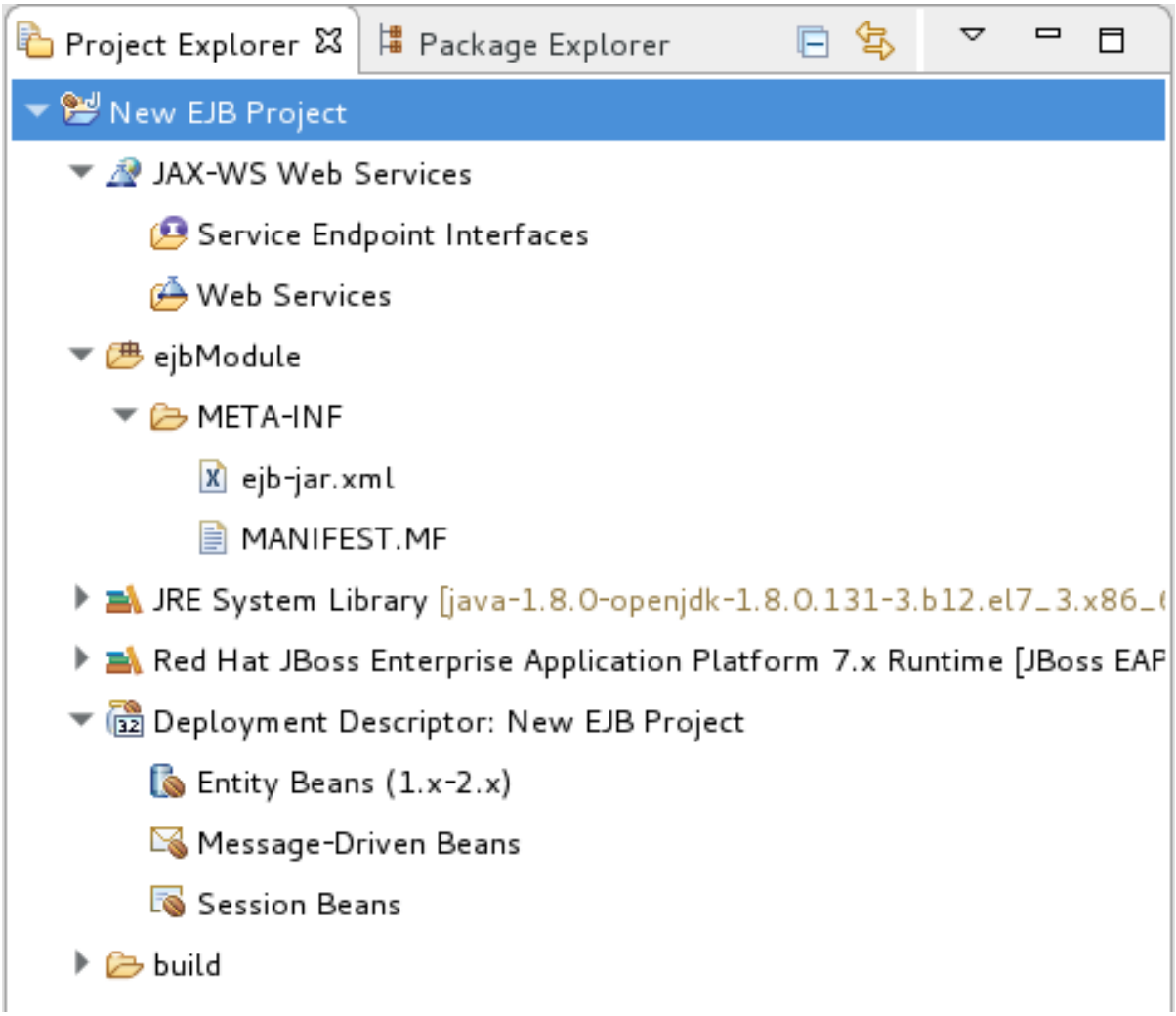

2. 以下の詳細を入力します。

- **Project name:** Red Hat CodeReady Studio に表示されるプロジェクトの名前、およびデプロイされた JAR ファイルのデフォルトのファイル名。
- **Project location:** プロジェクトファイルが保存されるディレクトリー。デフォルトは、現在のワークスペースのディレクトリーです。
- **Target runtime:** これは、プロジェクトに使用されるサーバーランタイムです。これは、デプロイするサーバーによって使用されるものと同じ **JBoss EAP** ランタイムに設定する必要があります。
- **EJB モジュールバージョン:** これは、エンタープライズ Bean が準拠する EJB 仕様のバージョンです。Red Hat は、**3.2** の使用を推奨します。
- **Configuration:** これにより、プロジェクトで対応している機能を調整することができます。選択したランタイムにデフォルト設定を使用します。
  **Next** をクリックして先に進みます。

3. **Java** プロジェクト設定画面では、Java ソースファイルを含むディレクトリーを追加し、ビルドの出力用にディレクトリーを指定できます。
   この設定は変更せず、**Next** をクリックします。

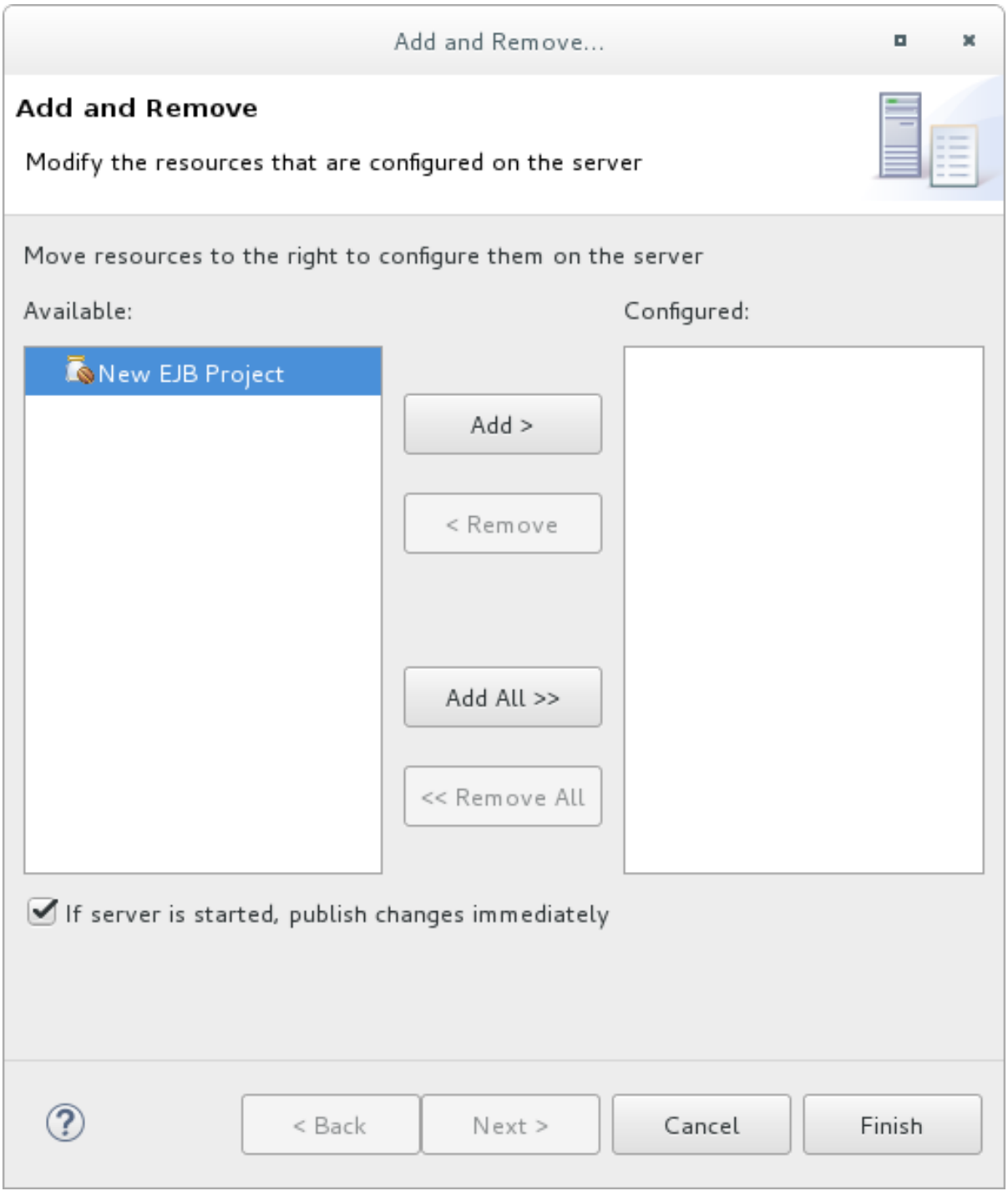
4. **EJB Module** 設定画面で、デプロイメント記述子が必要な場合に **ejb-jar.xml デプロイメント記述子の生成** を確認します。デプロイメント記述子は EJB 3.2 では任意で、必要な場合は後で追加できます。
   **Finish** をクリックするとプロジェクトが作成され、Project Explorer に表示されます。

**図2.2 Project Explorer で新たに作成された EJB プロジェクト**



5. デプロイメントのためにプロジェクトをサーバーに追加するには、**Servers** タブで対象のサーバーを右クリックし、**Add and Remove** を選択します。
**Add and Remove** ダイアログで、**Available** コラムからデプロイするリソースを選択して、**Add** ボタンをクリックします。リソースは **Configured** コラムに移動します。**Finish** をクリックしてダイアログを閉じます。

**図2.3 ダイアログの追加または削除**




これで、Red Hat CodeReady Studio には、指定のサーバーにビルドしてデプロイできる EJB プロジェクトができました。

**警告**

プロジェクトにエンタープライズ bean が追加されない場合、Red Hat CodeReady Studio は、**An EJB module must contain one or more enterprise beans**(EJB モジュールには、1 つ以上のエンタープライズ Bean が必要です) という警告が表示されます。この警告は、エンタープライズ Bean がプロジェクトに追加されると消えます。

## 2.2. MAVEN での EJB アーカイブプロジェクトの作成

このタスクは、JAR ファイルにパッケージ化されたエンタープライズ Bean を含む Maven を使用してプロジェクトを作成する方法を示しています。

**要件**

- Maven がすでにインストールされている。
- Maven の基本的な使用方法を理解している。

**Maven での EJB アーカイブプロジェクトの作成**

1. **Create the Maven project:**EJB プロジェクトは、Maven のアーキテクトシステムと **ejb-javaee7** アーキテクトタイプを使用して作成できます。これを実行するには、以下のパラメーターを指定して **mvn** コマンドを実行します。

```
$ mvn archetype:generate -DarchetypeGroupId=org.codehaus.mojo.archetypes -
DarchetypeArtifactId=ejb-javaee7
```

Maven はプロジェクトの **groupId**、**artifactId**、**version**、**package** をプロンプトします。

```
$ mvn archetype:generate -DarchetypeGroupId=org.codehaus.mojo.archetypes -
DarchetypeArtifactId=ejb-javaee7
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.0:generate (default-cli) @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.0:generate (default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.0:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype [org.codehaus.mojo.archetypes:ejb-javaee7:1.5] found in catalog remote
Define value for property 'groupId': : com.shinysparkly
Define value for property 'artifactId': : payment-arrangements
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  com.shinysparkly: :
Confirm properties configuration:
groupId: com.company
```

```
artifactId: payment-arrangements
version: 1.0-SNAPSHOT
package: com.company.collections
Y: :
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 32.440s
[INFO] Finished at: Mon Oct 31 10:11:12 EST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
[localhost]$
```

2. **Add your enterprise beans:**エンタープライズ Bean を作成し、Bean のパッケージの適切なサブディレクトリーにある **src/main/java** ディレクトリーの下のプロジェクトに追加します。

3. **Build the project:** プロジェクトを構築するには、**pom.xml** ファイルと同じディレクトリーで **mvn package** コマンドを実行します。これにより、Java クラスがコンパイルされ、JAR ファイルがパッケージ化されます。ビルド JAR ファイルの名前は **-.jar** です。このファイルは、**target/** ディレクトリーに置かれます。

これで、JAR ファイルをビルドし、パッケージ化する Maven プロジェクトが作成されました。これでプロジェクトにはエンタープライズ Bean を含めることができるようになりました。このプロジェクトは、JAR ファイルはアプリケーションサーバーにデプロイできます。

## 2.3. EJB プロジェクトを含む EAR プロジェクトの作成

このタスクでは、EJB プロジェクトを含む Red Hat CodeReady Studio で新しいエンタープライズアーカイブ (EAR) プロジェクトを作成する方法を説明します。

### 要件

- JBoss EAP のサーバーおよびサーバーランタイムが設定されている。

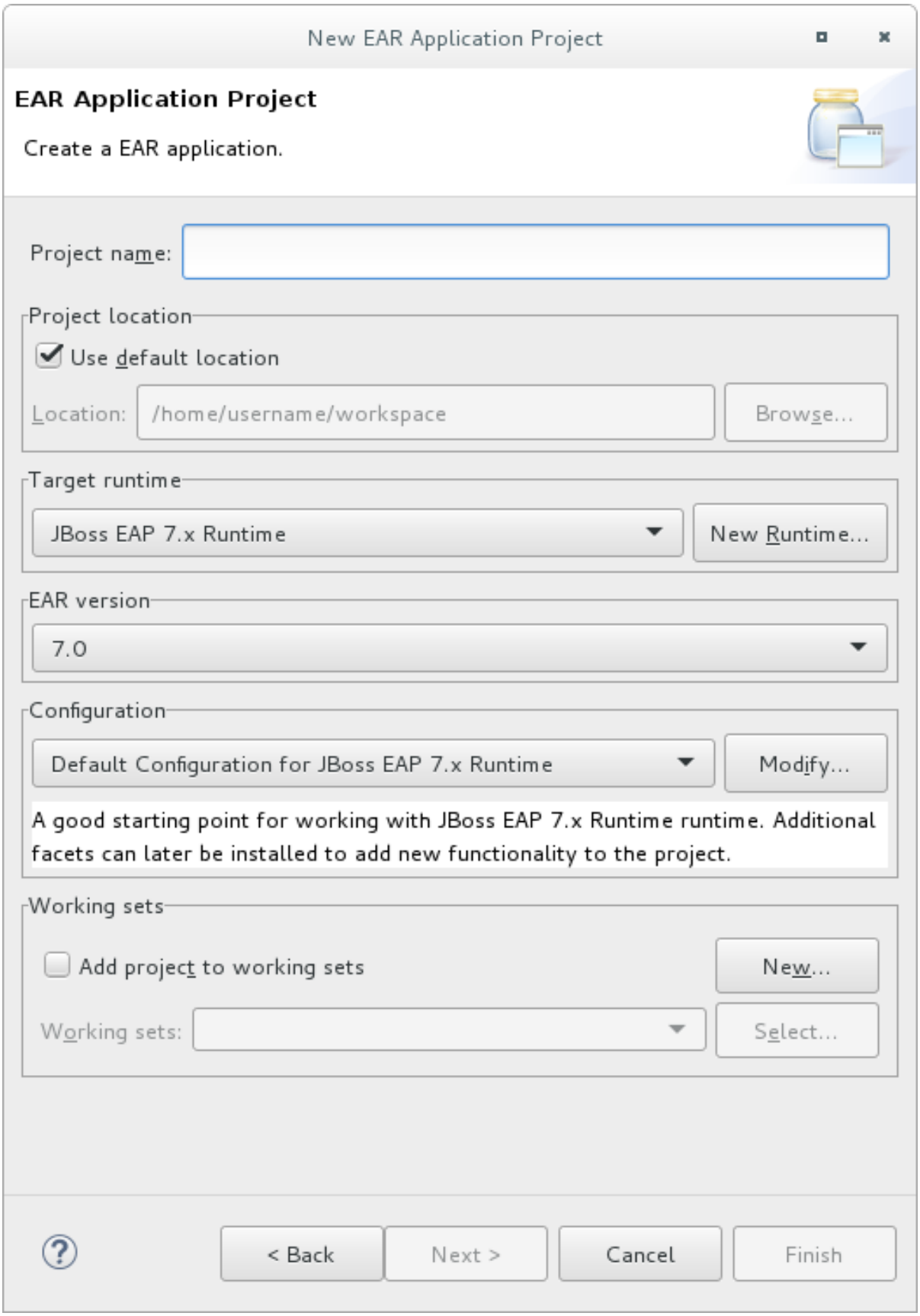
### EJB プロジェクトを含む EAR プロジェクトの作成

1. **New Java EE EAR Project** ウィザードを開きます。

   a. **File** メニューに移動して **New** を選択し、**Project** を選択します。

   b. **New Project** ウィザードが表示されたら、**Java EE/Enterprise Application Project** を選択し、**Next** をクリックします。

**図2.4 新規 EAR アプリケーションプロジェクトウィザード**




2. 以下の詳細を入力します。

- **Project name:** Red Hat CodeReady Studio に表示されるプロジェクトの名前、およびデプロイされた EAR ファイルのデフォルトのファイル名。

- **Project location:** プロジェクトファイルが保存されるディレクトリー。デフォルトは、現在のワークスペースのディレクトリーです。

- **Target runtime:** これは、プロジェクトに使用されるサーバーランタイムです。これは、デプロイするサーバーによって使用されるものと同じ JBoss EAP ランタイムに設定する必要があります。

- **Dear version:** これは、プロジェクトが準拠する Java EE 8 仕様のバージョンです。

- **Configuration:** これにより、プロジェクトで対応している機能を調整することができます。選択したランタイムにデフォルト設定を使用します。
**Next** をクリックして先に進みます。

1. 新しい EJB モジュールを追加します。
ウィザードの **Enterprise Applicaiton** ページから、新しいモジュールを追加できます。新しい EJB Project をモジュールとして追加するには、以下の手順に従います。

   a. **New Module**をクリックし、**Create Default Modules** チェックボックスのチェックを外し、**Enterprise Java Bean**を選択して **Next** をクリックします。**New EJB Project ウィザード** が表示されます。

   b. **New EJB Project** ウィザードは、新しいスタンドアロン EJB プロジェクトの作成に使用されるウィザードと同じで、新しい EJB プロジェクトウィザード で説明されています。
   プロジェクトの作成に必要な最低限の詳細は、以下のとおりです。

- プロジェクト名

- ターゲットランタイム

- EJB モジュールバージョン

- Configuration (設定)
ウィザードの他のすべての手順はオプションです。**Finish** をクリックし、EJB プロジェクトの作成を完了します。

新規作成した EJB プロジェクトは Java EE モジュール依存関係に一覧表示され、チェックボックスにチェックが付けられます。

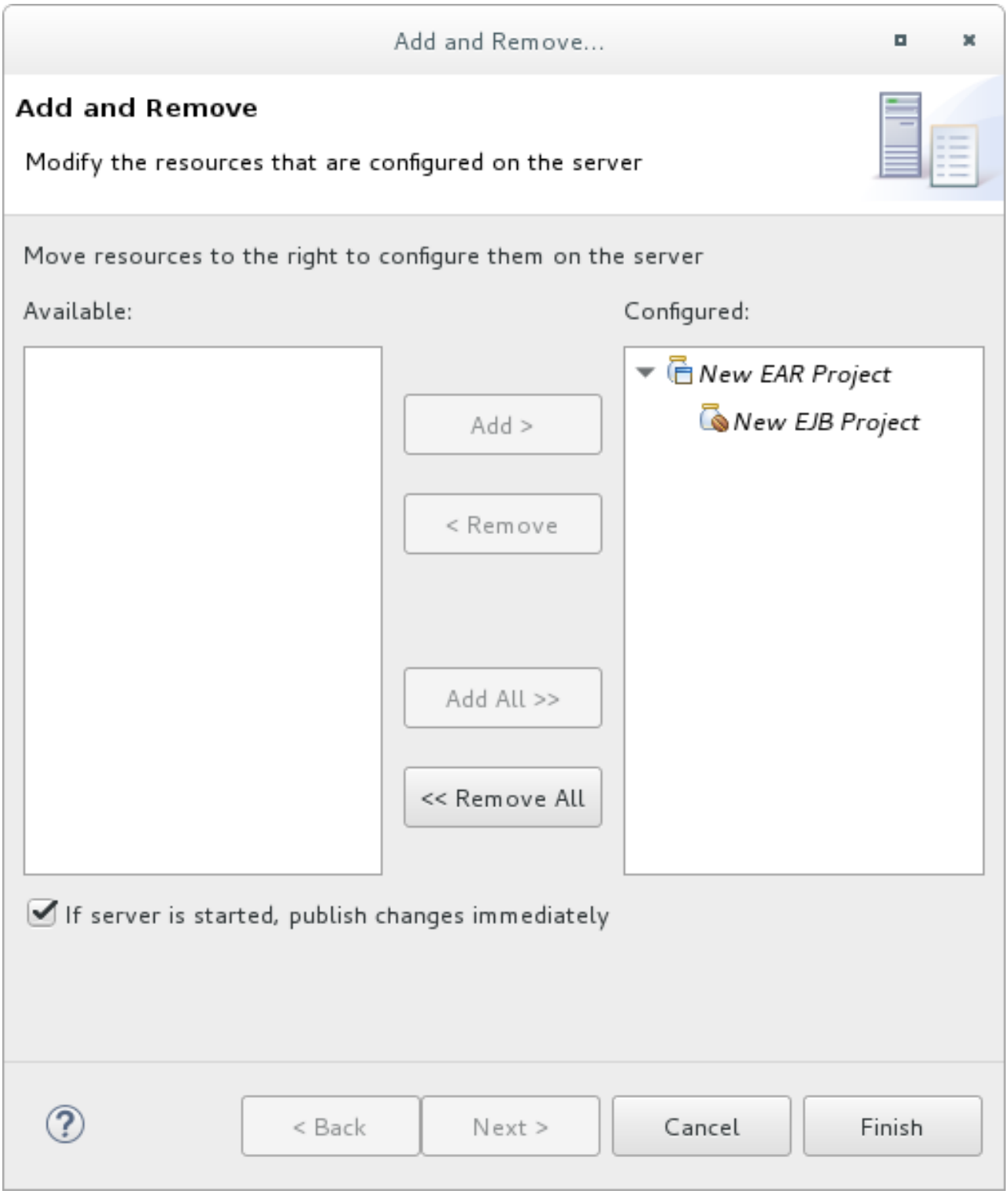
1. オプションで、**application.xml** デプロイメント記述子を追加します。
必要な場合は、**Generate application.xml deployment descriptor**チェックボックスにチェックを入れます。

2. **Finish** をクリックします。
EJB と EAR という 2 つのプロジェクトが表示されます。

3. デプロイメントを行うためにサーバーにビルドアーティファクトを追加します。
Server タブで、ビルドされたアーティファクトをデプロイするサーバーの **Servers** タブで **Add and Remove** ダイアログを開き、**Add and Remove** を選択します。

   **Available** コラムからデプロイする EAR リソースを選択し、**Add** ボタンをクリックします。リソースは Configured コラムに移動します。**Finish** をクリックしてダイアログを閉じます。

**図2.5 ダイアログの追加または削除**




これで、メンバー EJB プロジェクトを含む Enterprise Application Project ができました。これにより、EJB サブデプロイメントを含む単一の EAR デプロイメントとして指定のサーバーにビルドされ、デプロイされます。

## 2.4. EJB プロジェクトへのデプロイメント記述子の追加

EJB デプロイメント記述子は、記述子なしで作成された EJB プロジェクトに追加することができます。これを行うには、以下の手順に従います。

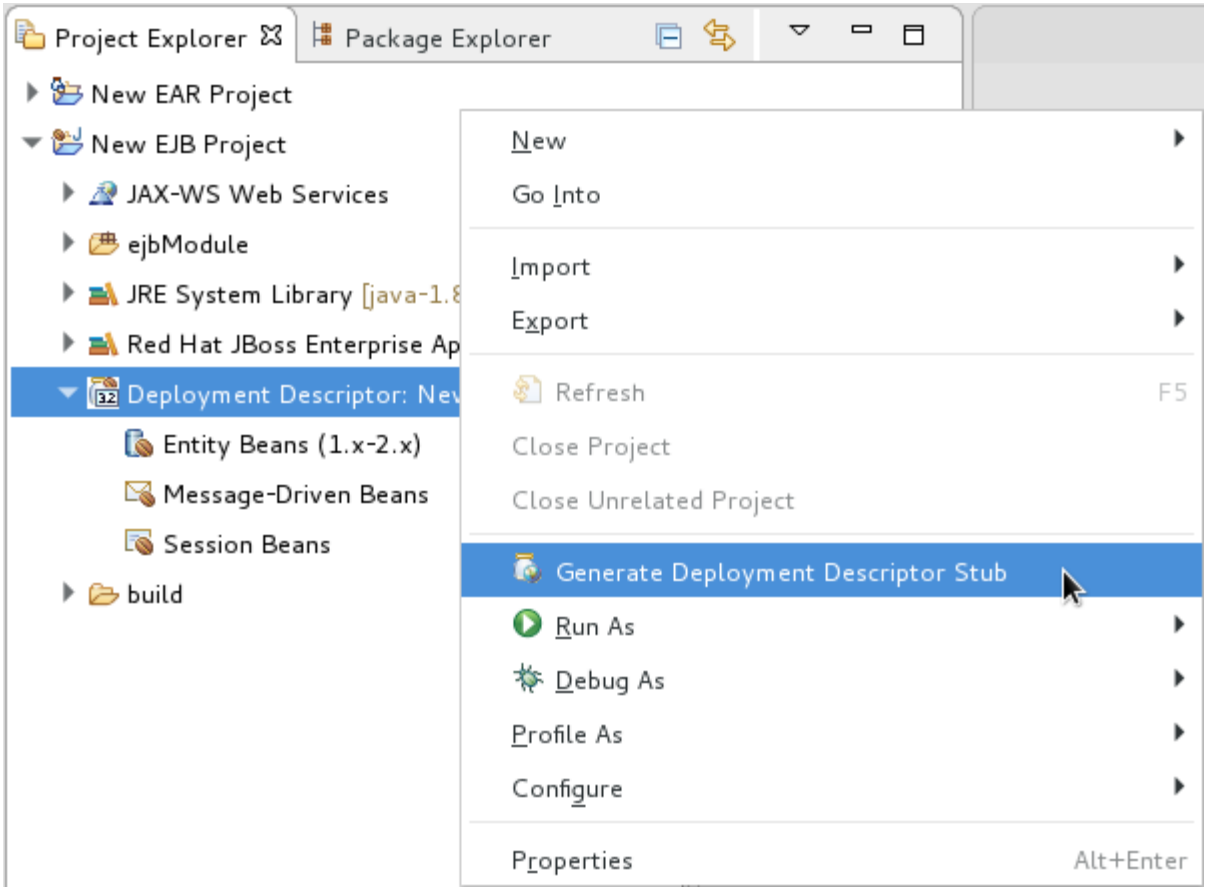
**要件**

- Red Hat CodeReady Studio には、EJB デプロイメント記述子を追加する EJB プロジェクトがあります。

### EJB プロジェクトへのデプロイメント記述子の追加

1. Red Hat CodeReady Studio でプロジェクトを開きます。
2. デプロイメント記述子を追加します。
   プロジェクトビューの **Deployment Descriptor** フォルダーを右クリックして、**Generate Deployment Descriptor** タブを選択します。

   **図2.6 デプロイメント記述子の追加**




   新しいファイル **ejb-jar.xml** が **ejbModule/META-INF/** に作成されます。プロジェクトビューの **Deployment Descriptor** フォルダーをダブルクリックして、このファイルを開きます。

# 第3章 セッション BEAN

## 3.1. セッション BEAN

セッション Bean は、関連する一連のビジネスプロセスまたはタスクをカプセル化し、それらを要求するクラスにインジェクトするエンタープライズ Bean です。セッション bean のタイプには、ステートレス、ステートフル、シングルトンの 3 つがあります。

## 3.2. ステートレスセッション BEAN

ステートレスセッション Bean は、最も広く使用されているセッション Bean のタイプです。クライアントアプリケーションにビジネスメソッドを提供しますが、メソッド呼び出し間の状態を維持しません。各メソッドは、セッション Bean 内の共有状態に依存しない完全なタスクです。状態がないため、アプリケーションサーバーでは、各メソッド呼び出しが同じインスタンスで実行されるようにする必要はありません。これにより、ステートレスセッション Bean は非常に効率的でスケーラブルなものになります。

## 3.3. ステートフルセッション BEAN

ステートフルセッション Bean は、ビジネスメソッドをクライアントアプリケーションに提供するエンタープライズ Bean で、クライアントとの会話状態を維持します。これは、複数の手順で実行する必要があるタスク、またはメソッド呼び出しに使用してください。これらの各タスクは、維持される前のステップの状態に依存します。アプリケーションサーバーは、各クライアントが各メソッド呼び出しに対してステートフルセッション Bean の同じインスタンスを受け取るようにします。

## 3.4. シングルトンセッション BEAN

シングルトンセッション Bean はアプリケーションごとにインスタンス化されるセッション Bean で、シングルトン bean に対する各クライアントリクエストは同じインスタンスに送信されます。シングルトン bean は、1994 年に出版された Erich Gamma、Richard Helm、Ralph Jonson、John Vlissides 著の**Design Patterns: Elements of Reusable Object-Oriented Software**で説明されている Singleton Design Pattern の実装です。

シングルトン bean は、すべてのセッション bean タイプのメモリーフットプリントを最小限に抑えますが、スレッドセーフとして設計する必要があります。EJB 3.2 では、開発者がスレッドセーフシングルトン Bean を簡単に実装できるように、コンテナー管理コンカレンシー (デプロイメント) を利用できます。ただし、CMC が詳細な柔軟性を提供しない場合、従来のマルチスレッドコード (Bean-managed Concurrency または BMC) を使用してシングルトン bean を書き込むこともできます。

## 3.5. RED HAT CODEREADY STUDIO でプロジェクトにセッション BEAN を追加する

Red Hat CodeReady Studio には、エンタープライズ bean クラスを迅速に作成するのに使用できるいくつかのウィザードがあります。以下の手順では、Red Hat CodeReady Studio ウィザードを使用してセッション bean をプロジェクトに追加する方法を示します。

**要件**

- 1 つ以上のセッション Bean を追加する Red Hat CodeReady Studio に EJB または動的 Web プロジェクトがある。

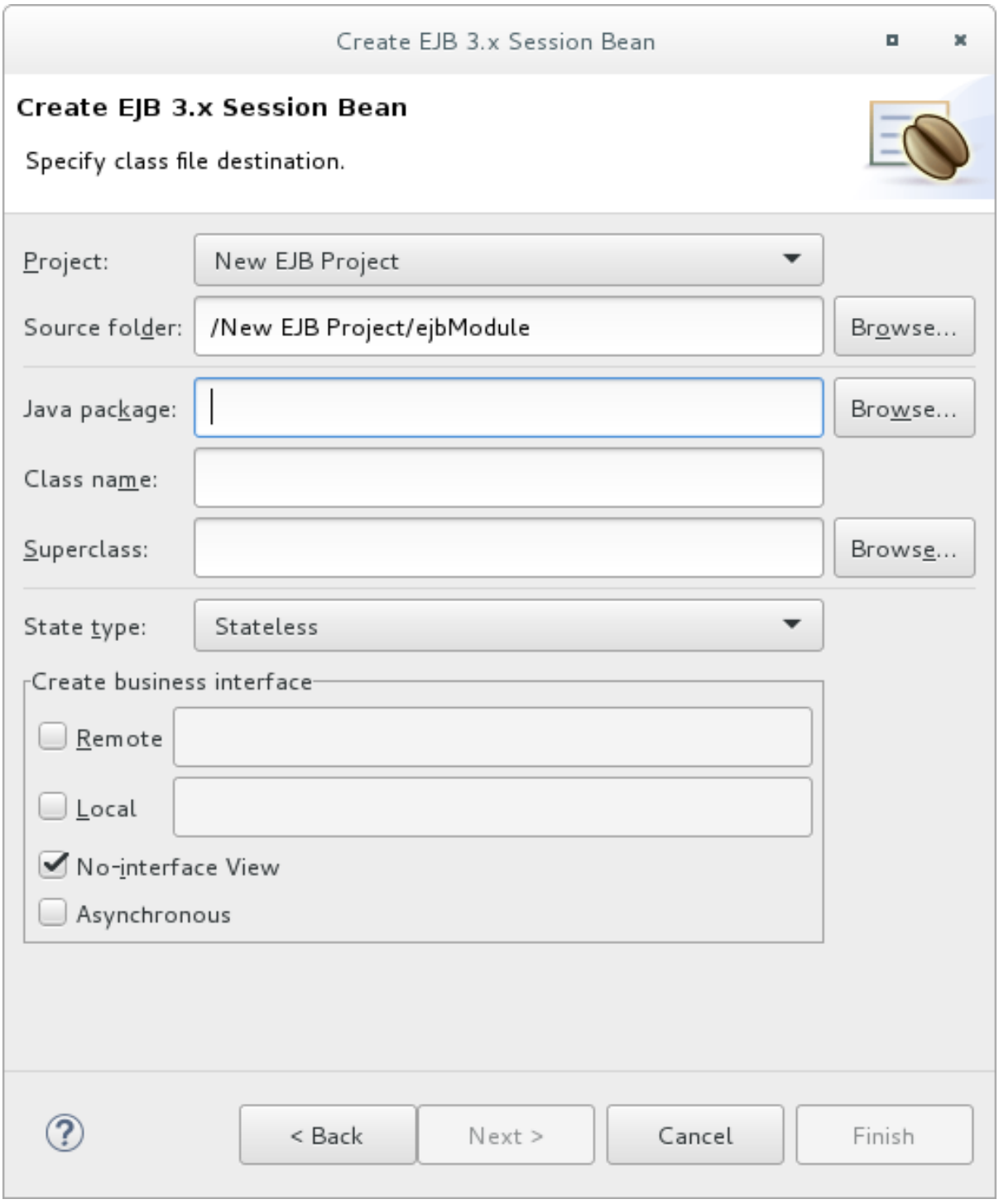
**Red Hat CodeReady Studio でプロジェクトにセッション Bean を追加する**

1. Red Hat CodeReady Studio でプロジェクトを開きます。

1. Red Hat CodeReady Studio でプロジェクトを開きます。

2. Create EJB 3.x Session Bean ウィザードを開きます。
**Create EJB 3.x Session Bean**ウィザードを開くには、**File** メニューに移動し、**New** を選択して **Session Bean (EJB 3.x)** を選択します。

**図3.1 Create EJB 3.x Session Bean ウィザード**




3. 以下の詳細を指定します。

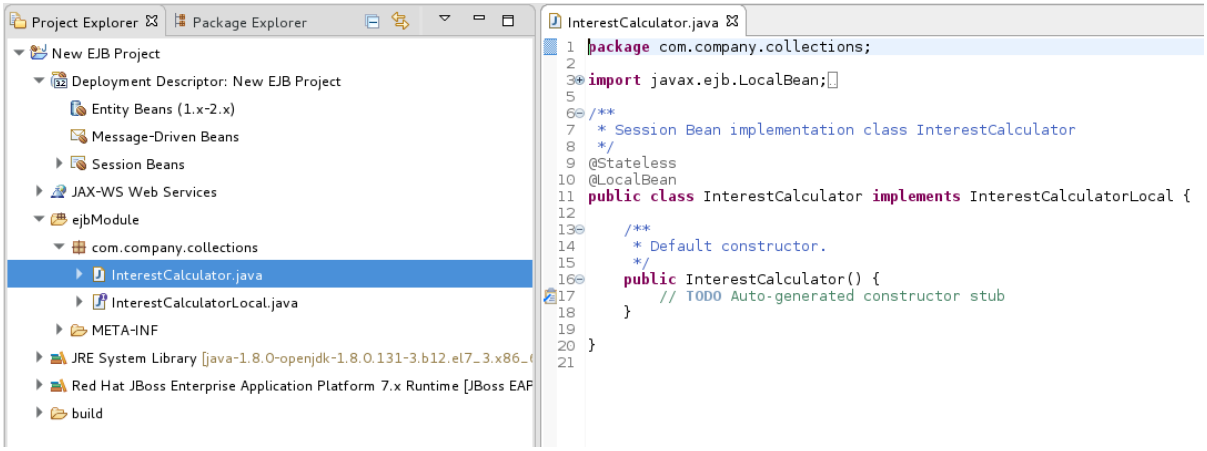
- **Project:** 正しいプロジェクトが選択されていることを確認します。
- **Source folder:** これは、Java ソースファイルが作成されるフォルダーです。通常、これは変更する必要はありません。
- **Package:** クラスが属するパッケージを指定します。

- **Class name:** セッション Bean になるクラスの名前を指定します。
- **Superclass:** セッション Bean クラスはスーパークラスから継承できます。セッションにスーパークラスがある場合は、これを指定します。
- **State type:** セッション Bean の状態タイプ (stateless、stateful、または singleton) を指定します。
- **Business interfaces:** デフォルトでは、**No-interface** ボックスがチェックされるため、インターフェイスは作成されません。定義するインターフェイスのボックスにチェックを入れ、必要に応じて名前を調整します。
  Web アーカイブ (WAR) のエンタープライズ Bean は EJB 3.2 Lite のみをサポートし、これにはリモートビジネスインターフェイスが含まれないことに注意してください。

  **Next** をクリックします。

4. ここで追加情報を入力して、セッション Bean をさらにカスタマイズできます。ここで情報を変更する必要はありません。
   変更可能なアイテムは次のとおりです。
   - Bean 名
   - マップされた名前
   - トランザクションタイプ (管理対象コンテナーまたは管理対象 Bean)
   - Bean が実装する必要のある追加のインターフェイスを指定可能
   - EJB 2.x Home and Component インターフェイスを必要に応じて指定することもできます。
5. **Finish** をクリックすると、新しいセッション Bean が作成され、プロジェクトに追加されます。新しいビジネスインターフェイスのファイルが指定されていれば、それらも作成されます。

**図3.2 Red Hat CodeReady Studio の新しいセッション Bean**



# 第4章 メッセージ駆動 BEAN

## 4.1. メッセージ駆動 BEAN

メッセージ駆動 Bean (MDB) は、アプリケーション開発のイベント駆動モデルを提供します。MDB のメソッドは、クライアントコードにインジェクトされず、クライアントコードから呼び出しませんが、Jakarta Messaging サーバーなどのメッセージングサービスからのメッセージの受信によってトリガーされます。Jakarta EE 仕様では Jakarta Messaging に対応する必要がありますが、他のメッセージングシステムも同様にサポートすることができます。

MDB は、特別なタイプのステートレスセッション Bean です。**onMessage (Message message)** というメソッドを実装します。この方法は、MDB リッスンしている Jakarta Messaging の宛先がメッセージを受信するとトリガーされます。つまり、MDB は Jakarta Messaging プロバイダーからのメッセージを受信してトリガーされます。通常、メソッドが EJB クライアントによって呼び出されるステートレスセッション Bean とは異なります。

MDB は非同期にメッセージを処理します。デフォルトでは、各 MDB は最大 16 のセッションを持つことができ、そこで各セッションがメッセージを処理できます。メッセージの順番の保証はありません。メッセージの順序付けを行うには、MDB のセッションプールを **1** に制限する必要があります。

**例: セッションプールを 1 に設定する管理 CLI コマンド**

```
/subsystem=ejb3/strict-max-bean-instance-pool=mdb-strict-max-pool:write-attribute(name=derive-size,value=undefined)

/subsystem=ejb3/strict-max-bean-instance-pool=mdb-strict-max-pool:write-attribute(name=max-pool-size,value=1)

reload
```

## 4.2. メッセージ駆動 BEAN が制御する配信

JBoss EAP は、特定の MDB でメッセージのアクティブな受信を制御する以下の属性を提供します。

- Delivery Active
- 配信グループ
- Clustered Singleton MDBs

### 4.2.1. Delivery Active

メッセージ駆動 Bean (MDB) の配信アクティブ (Delivery Active) 設定は、MDB がメッセージを受信しているかどうかを示します。MDB がメッセージを受信していない場合、メッセージはトピックまたはキューのルールに従ってキューまたはトピックに保存されます。

XML またはアノテーションを使用して **delivery-group** の **active** 属性を設定できます。また、管理 CLI を使用してデプロイメント後にこの値を変更できます。デフォルトでは、**active** 属性が有効になり、MDB がデプロイされるとすぐにメッセージの配信が行われます。

**Jboss-ejb3.xml ファイルでの Delivery Active の設定**
**Jboss-ejb3.xml** ファイルで **active** の値を **false** に設定し、MDB がデプロイされた直後にメッセージを受信しないことを示します。

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:d="urn:delivery-active:1.1"
    xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee http://www.jboss.org/j2ee/schema/jboss-
ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
    version="3.1"
    impl-version="2.0">
    <assembly-descriptor>
        <d:delivery>
            <ejb-name>HelloWorldQueueMDB</ejb-name>
            <d:active>false</d:active>
        </d:delivery>
    </assembly-descriptor>
</jboss:ejb-jar>
```

アプリケーションのすべての MDB にアクティブな値を適用する場合は、**ejb-name** の代わりにワイルドカード * を使用できます。

**アノテーションを使用した Delivery Active の設定**
**org.jboss.ejb3.annotation.DeliveryActive** アノテーションを使用することもできます。例を以下に示します。

```
@MessageDriven(name = "HelloWorldMDB", activationConfig = {
 @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
 @ActivationConfigProperty(propertyName = "destination", propertyValue =
"queue/HELLOWORLDMDBQueue"),
 @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-
acknowledge") })
@DeliveryActive(false)

public class HelloWorldMDB implements MessageListener {
    public void onMessage(Message rcvMessage) {
      // ...
    }
}
```

Maven を使用してプロジェクトをビルドする場合は、以下の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.ejb3</groupId>
    <artifactId>jboss-ejb3-ext-api</artifactId>
    <version>2.2.0.Final</version>
</dependency>
```

**管理 CLI を使用した配信アクティブの設定**
管理 CLI を使用して、デプロイメント後に **delivery-group** の **active** 属性を設定できます。これらの管理操作は、**active** 属性の値を動的に変更し、MDB の配信を有効または無効にします。Delivery Actgive の値を変更するこの方法は、サーバーを再起動しても維持されません。実行時に、管理するインスタンスに接続し、配信を管理する MDB のパスを入力します。例を以下に示します。

- 管理するインスタンスに移動します。

```
cd deployment=helloworld-mdb.war/subsystem=ejb3/message-driven-
bean=HelloWorldQueueMDB
```

- MDB への配信を停止するには、以下を行います。

```
:stop-delivery
```

- MDB への配信を開始するには、以下を行います。

```
:start-delivery
```

**MDB Delivery Active ステータスの表示**
管理コンソールを使用して、MDB の現在の Delivery Active ステータスを表示できます。

1. **Runtime** タブを選択し、該当するサーバーを選択します。
2. **EJB** をクリックし、**HelloWorldQueueMDB** などの子リソースを選択します。

**結果**

ステータスは **Delivery Active: true** または **Delivery Active: false** になります。

### 4.2.2. 配信グループ

配信グループは、MDB のグループの **delivery-active** ステータスを管理する方法を提供します。MDB は、複数の 配信グループに所属できます。メッセージ配信は、MDB が属するすべての配信グループがアクティブの場合にのみ有効になります。クラスター化されたシングルトン MDB の場合、メッセージ配信は、MDB に関連付けられたすべての配信グループがアクティブである場合にのみ、クラスターのシングルトンノードでのみアクティブになります。

XML 設定または管理 CLI のいずれかを使用して **ejb3** サブシステムに配信グループを追加できます。

**jboss-ejb3.xml ファイルでの配信グループの設定**

```
<delivery>
  <ejb-name>MdbName<ejb-name>
  <delivery-group>passive</delivery-group>
</delivery>
```

サーバー側では、以下の例のように **active** 属性を **true** に設定して **delivery-groups** を有効にしたり、**active** 属性を **false** に設定して無効にすることができます。

```
<delivery-groups>
  <delivery-group name="group" active="true"/>
</delivery-groups>
```

**管理 CLI を使用した配信グループの設定**
**delivery-groups** の状態は、管理 CLI を使用して更新できます。例を以下に示します。

```
/subsystem=ejb3/mdb-delivery-group=group:add
/subsystem=ejb3/mdb-delivery-group=group:remove
/subsystem=ejb3/mdb-delivery-group=group:write-attribute(name=active,value=true)
```

**jboss-ejb3.xml** ファイルで Delivery Active を設定するか、アノテーションを使用すると、サーバーを再起動しても保持されます。ただし、管理 CLI を使用して配信を停止または開始すると、サーバーの再起動時に維持されません。

**アノテーションを使用した複数の配信グループの設定**
グループに属する各 MDB クラスで **org.jboss.ejb3.annotation.DeliveryGroup** アノテーションを使用できます。

```
@MessageDriven(name = "HelloWorldQueueMDB", activationConfig = {
        @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Queue"),
        @ActivationConfigProperty(propertyName = "destination", propertyValue =
"queue/HELLOWORLDMDBQueue"),
        @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-
acknowledge") })
@DeliveryGroup("delivery-group-1")
@DeliveryGroup("delivery-group-2")
public class HelloWorldQueueMDB implements MessageListener {
...
}
```

### 4.2.3. Clustered Singleton MDBs

MDB がクラスター化されたシングルトンとして識別され、クラスターにデプロイされると、単一のノードのみがアクティブになります。このノードは、メッセージを順次使用できます。サーバーノードが失敗すると、クラスター化されたシングルトン MDB からのアクティブなノードがメッセージを消費し始めます。

**MDB をクラスター化されたシングルトンとして特定**
以下のいずれかの手順を使用して、MDB をクラスター化シングルトンとして識別できます。

- 以下の例のように、clustered-singleton XML 要素を使用します。

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:c="urn:clustering:1.1"
               xmlns:d="urn:delivery-active:1.2"
               xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
               version="3.1"
               impl-version="2.0">
   <assembly-descriptor>
      <c:clustering>
         <ejb-name>HelloWorldQueueMDB</ejb-name>
         <c:clustered-singleton>true</c:clustered-singleton>
      </c:clustering>
       <d:delivery>
            <ejb-name>*</ejb-name>
            <d:group>delivery-group-1</d:group>
            <d:group>delivery-group-2</d:group>
```

```
        </d:delivery>
    </assembly-descriptor>
</jboss:ejb-jar>
```

- MDB クラスで **@org.jboss.ejb3.annotation.ClusteredSingleton** を使用します。この手順では、サーバーに追加の設定は必要ありません。クラスター環境でサービスを実行する必要があります。

**注記**

**singleton master** に選択されているクラスターノードがわからないため、クラスター全体 (具体的にはクラスターのすべてのノード) で **delivery-group** を有効化する必要があります。サーバーが **singleton master** となるノードを選択し、そのノードに必要な **delivery-group** をアクティブにしていない場合、クラスター内のノードはメッセージを受信しません。

JBoss EAP に含まれる **messaging-clustering-singleton** クイックスタートは、統合された Apache ActiveMQ Artemis でのクラスターリングの使用を示しています。これは **helloworld-mdb** クイックスタートと同じソースコードを使用しますが、設定にのみ違いがあり、クラスター化されたシングルトンとして実行されます。このクイックスタートには、2 つの Jakarta Messaging リソースが含まれています。

- Java Naming and Directory Interface にバインドされる **HELLOWORLDMDBQueue** という名前のキュー (**java:/queue/HELLOWORLDMDBQueue)**
- Java Naming and Directory Interface を **java:/topic/HELLOWORLDMDBTopic** としてバインドする **HELLOWORLDMDBTopic** という名前のトピック

いずれも **jboss-ejb3.xml** ファイルで指定されるシングルトン設定を含みます。

```
<c:clustering>
    <ejb-name>*</ejb-name>
    <c:clustered-singleton>true</c:clustered-singleton>
</c:clustering>
```

**<ejb-name>** 要素のワイルドカードアスタリスク * は、アプリケーションに含まれるすべての MDB がクラスター化シングルトンであることを示しています。これにより、クラスター内の単一のノードのみが、特定の時点で MDB がアクティブになります。このアクティブなノードがシャットダウンすると、クラスター内の別のノードが MDB でアクティブなノードになり、シングルトンプロバイダーになります。

**jboss-ejb3.xml** ファイルに、配信グループの設定を検索することもできます。

```
<d:delivery>
    <ejb-name>HelloWorldTopicMDB</ejb-name>
    <d:group>my-mdb-delivery-group</d:group>
</d:delivery>
```

この場合、MDB のいずれかの **HelloWorldTopicMDB** のみが配信グループに関連付けられます。MDB が使用するすべて配信グループは、**ejb3** サブシステム設定で設定する必要があります。配信グループは有効または無効にできます。クラスターノードで配信グループが無効になっていると、その配信グループに属するすべての MDB が、対応するクラスターノードで非アクティブになります。非クラスター化環境で配信グループを使用する場合は、配信グループが有効になっているたびに MDB がアクティブになります。

配信グループがシングルトンプロバイダーとともに使用されている場合、そのノードで配信グループが有効になっている場合にのみ、MDB をシングルトンプロバイダーノードでアクティブにすることができます。それ以外の場合、MDB はそのノードで非アクティブになり、クラスターの他のすべてのノードは非アクティブになります。

メッセージングクラスターリング用にサーバーを設定する方法や、コード例を確認する方法については、このクイックスタートに含まれている **README.html** ファイルを参照してください。

JBoss EAP クイックスタートのダウンロードおよび使用方法の詳細は、JBoss EAPJBoss EAP **Getting Started Guide**の Using the Quickstart Examples を参照してください。

## 4.3. RED HAT CODEREADY STUDIO での JAKARTA MESSAGING ベースのメッセージ駆動 BEAN の作成

この手順では、Red Hat CodeReady Studio で Jakarta Messaging ベースのメッセージ駆動 Bean をプロジェクトに追加する方法を説明します。この手順では、アノテーションを使用する EJB 3.x メッセージ駆動 Bean を作成します。

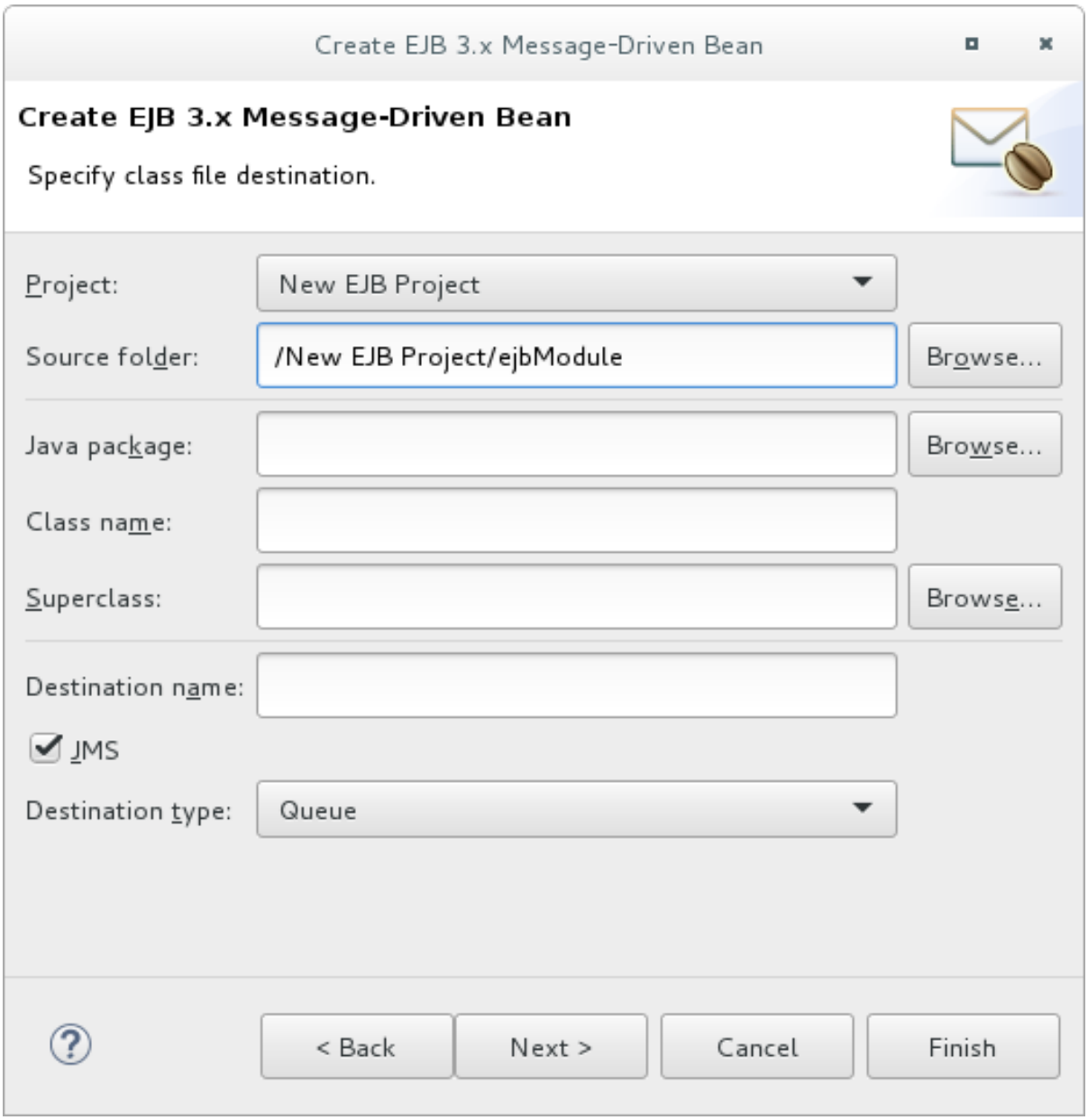
**要件**

- Red Hat CodeReady Studio で既存のプロジェクトを開いておく必要があります。
- Bean がリッスンする Jakarta Messaging 宛先の名前とタイプを知っている必要があります。
- Jakarta Messaging のサポートは、この Bean がデプロイされる JBoss EAP 設定で有効にする必要があります。

**Red Hat CodeReady Studio での Jakarta Messaging ベースのメッセージ駆動 Bean の追加**

1. **Create EJB 3.x Message-LoadBalancern Bean**ウィザードを開きます。
   **File → New → Other**の順に移動します。**EJB/Message-Driven Bean (EJB 3.x)** を選択し、**Next** ボタンをクリックします。

**図4.1 Create EJB 3.x Message-Driven Bean ウィザード**




2. クラスファイル宛先の詳細を指定します。
ここでは、bean クラスに指定する詳細セット (プロジェクト、Java クラス、メッセージの宛先) があります。

- **Project:**
  - ワークスペースに複数のプロジェクトが存在する場合は、**Project** メニューで適切なプロジェクトが選択されていることを確認します。
  - 新規 Bean のソースファイルが作成されるフォルダーは、選択したプロジェクトのディレクトリーの下の **ejbModule** になります。これは、特定の要件がある場合にのみ変更します。
- **Java Class:**
  - 必須フィールドは **Java package** と **Class name** です。
  - アプリケーションのビジネスロジックで必要な場合を除いて、スーパークラスを指定する必要はありません。

- **Message Destination:**
  - 以下は、Jakarta Messaging ベースのメッセージ駆動 Bean に提供する必要のある詳細です。
    - **Destination name**: Bean が応答するメッセージが含まれるキューまたはトピック名。
    - デフォルトでは、**JMS** チェックボックスが選択されます。これは変更しないでください。
    - 必要に応じて **Destination type** を **Queue** または **Topic** に設定します。
      **Next** ボタンをクリックしてください。

3. メッセージ駆動 Bean 固有の情報を入力します。
   ここでのデフォルト値は、コンテナー管理トランザクションを使用する Jakarta Messaging ベースのメッセージ駆動 Bean に適しています。
   - Bean が Bean 管理対象トランザクションを使用する場合は、**Transaction type** を Bean に変更します。
   - クラス名とは異なる Bean 名が必要な場合は **Bean name** を変更します。
   - **JMS Message Listener** インターフェイスはすでにリストされています。インターフェイスがアプリケーションのビジネスロジックに固有のものでない場合は、インターフェイスを追加または削除する必要はありません。
   - メソッドスタブを作成するためのチェックボックスは選択したままにしておきます。
     **Finish** ボタンをクリックします。

**結果**

メッセージ駆動 Bean は、デフォルトのコンストラクターと **onMessage()** メソッド用のスタブメソッドで作成されます。Red Hat CodeReady Studio エディターウィンドウが開き、対応するファイルが表示されます。

## 4.4. MDB に対して JBOSS-EJB3.XML でのリソースアダプターの指定

**jboss-ejb3.xml** デプロイメント記述子では、MDB が使用するリソースアダプターを指定できます。

MDB に対して **jboss-ejb3.xml** でリソースアダプターを指定するには、以下の例を使用します。

**例: MDB リソースアダプターの jboss-ejb3.xml 設定**

```
<jboss xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:jee="http://java.sun.com/xml/ns/javaee"
  xmlns:mdb="urn:resource-adapter-binding">
  <jee:assembly-descriptor>
    <mdb:resource-adapter-binding>
      <jee:ejb-name>MyMDB</jee:ejb-name>
      <mdb:resource-adapter-name>MyResourceAdapter.rar</mdb:resource-adapter-name>
    </mdb:resource-adapter-binding>
  </jee:assembly-descriptor>
</jboss>
```

EAR にあるリソースアダプターでは、**<mdb:resource-adapter-name>** に以下の構文を使用する必要があります。

- 別の EAR にあるリソースアダプターの場合:

```
<mdb:resource-adapter-name>OtherDeployment.ear#MyResourceAdapter.rar</mdb:resource-adapter-name>
```

- MDB と同じ EAR にあるリソースアダプターでは、EAR 名を省略できます。

```
<mdb:resource-adapter-name>#MyResourceAdapter.rar</mdb:resource-adapter-name>
```

## 4.5. MDB のリソース定義アノテーションを使用したクラスターへのデプロイ

**@JMSConnectionFactoryDefinition** および **@JMSDestinationDefinition** アノテーションを使用して、メッセージ駆動 Bean の接続ファクトリーと宛先を作成する場合は、オブジェクトが MDB がデプロイされたサーバーにのみ作成されることに注意してください。MDB もクラスター内のすべてのノードにデプロイされない限り、クラスターのすべてのノードには作成されません。これらのアノテーションで設定されたオブジェクトは MDB がデプロイされているサーバー上でのみ作成されるため、MDB がリモートサーバーからメッセージを読み取り、リモートサーバーに送信するリモート Jakarta Connectors トポロジーに影響します。

## 4.6. アプリケーションで EJB および MDB プロパティーの置換を有効にする

Red Hat JBoss Enterprise Application Platform では、**@ActivationConfigProperty** and **@Resource** アノテーションを使用して EJB および MDB でプロパティー置換を有効にできます。プロパティーの置換には、以下の設定およびコードの変更が必要です。

- JBoss EAP サーバー設定ファイルでプロパティー置換を有効にする必要があります。
- JBoss EAP サーバーの起動時に 、 システムプロパティーをサーバー設定ファイルに定義するか、引数として渡す必要があります。
- 置換変数を使用するように アプリケーションコードを変更する 必要があります。

以下の例は、プロパティー置換を使用するために JBoss EAP に同梱される **helloworld-mdb** クイックスタートを変更する方法を示しています。完全な作業例は、**helloworld-mdb-propertysubstitution** クイックスタートを参照してください。

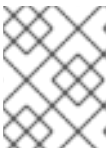
### 4.6.1. プロパティー置換を有効にするようにサーバーを設定

JBoss EAP サーバーでプロパティーの置換を有効にするには、サーバー設定の **ee** サブシステムの **annotation-property-replacement** 属性を **true** に設定する必要があります。

1. サーバー設定ファイルをバックアップします。
   **helloworld-mdb-propertysubstitution** クイックスタートの例では、スタンドアロンサーバーの完全なプロファイルが必要であるため、これは **EAP_HOME/standalone/configuration/standalone-full.xml** ファイルになります。管理対象ドメインでサーバーを実行している場合は、**EAP_HOME/domain/configuration/domain.xml** ファイルになります。

2. JBoss EAP のインストールディレクトリーへ移動し、フルプロファイルでサーバーを起動します。

```
$ EAP_HOME/bin/standalone.sh -c standalone-full.xml
```

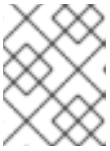


**注記**

Windows Server の場合は、**EAP_HOME\bin\standalone.bat** スクリプトを使用します。

3. 管理 CLI を起動します。

```
$ EAP_HOME/bin/jboss-cli.sh --connect
```



**注記**

Windows Server の場合は、**EAP_HOME\bin\jboss-cli.bat** スクリプトを使用します。

4. アノテーションプロパティーの置換を有効にするには、以下のコマンドを実行します。

```
/subsystem=ee:write-attribute(name=annotation-property-replacement,value=true)
```

以下の結果が表示されるはずです。

```
{"outcome" => "success"}
```

5. JBoss EAP サーバー設定ファイルへの変更を確認します。**ee** サブシステムには以下の XML が含まれるはずです。

**ee サブシステムの設定例**

```
<subsystem xmlns="urn:jboss:domain:ee:4.0">
  ...
  <annotation-property-replacement>true</annotation-property-replacement>
  ...
</subsystem>
```

### 4.6.2. システムプロパティーの定義

サーバー設定ファイルでシステムプロパティーを指定するか、JBoss EAP サーバーの起動時にコマンドラインの引数としてプロパティーを渡すことができます。サーバー設定ファイルで定義されたシステムプロパティーは、サーバーの起動時にコマンドラインに渡されるプロパティーよりも優先されます。

#### 4.6.2.1. サーバー設定でのシステムプロパティーの定義

1. 管理 CLI を起動します。
2. 以下のコマンド構文を使用して、JBoss EAP サーバーのシステムプロパティーを設定します。

**システムプロパティーを追加する構文**

```
-
```

```
/system-property=PROPERTY_NAME:add(value=PROPERTY_VALUE)
```

以下のシステムプロパティーは、**helloworld-mdb-propertysubstitution** クイックスタートに設定されています。

**システムプロパティーを追加するコマンドの例**

```
/system-property=property.helloworldmdb.queue:add(value=java:/queue/HELLOWORLDMDBPropQueue)
/system-property=property.helloworldmdb.topic:add(value=java:/topic/HELLOWORLDMDBPropTopic)
/system-property=property.connection.factory:add(value=java:/ConnectionFactory)
```

3. JBoss EAP サーバー設定ファイルへの変更を確認します。以下のシステムプロパティーは、**<extensions>** の後に表示されるはずです。

**システムプロパティーの設定例**

```
<system-properties>
    <property name="property.helloworldmdb.queue"
value="java:/queue/HELLOWORLDMDBPropQueue"/>
    <property name="property.helloworldmdb.topic"
value="java:/topic/HELLOWORLDMDBPropTopic"/>
    <property name="property.connection.factory" value="java:/ConnectionFactory"/>
</system-properties>
```

#### 4.6.2.2. サーバー起動時にシステムプロパティーを引数として渡す

必要に応じて、**-DPROPERTY_NAME=PROPERTY_VALUE** の形式で JBoss EAP サーバーを起動したときに、コマンドラインで引数を渡すことができます。以下は、前のセクションで定義されたシステムプロパティーの引数を渡す方法の例になります。

**サーバー起動コマンドによりシステムプロパティーを渡す例**

```
$ EAP_HOME/bin/standalone.sh -c standalone-full.xml  -
Dproperty.helloworldmdb.queue=java:/queue/HELLOWORLDMDBPropQueue  -
Dproperty.helloworldmdb.topic=java:/topic/HELLOWORLDMDBPropTopic  -
Dproperty.connection.factory=java:/ConnectionFactory
```

### 4.6.3. システムプロパティーの置換を使用するようアプリケーションコードを変更します。

ハードコーディングされた **@ActivationConfigProperty** および **@Resource** アノテーションの値を、新たに定義したシステムプロパティーの置換値に置き換えます。以下は、新たに定義したシステムプロパティーの置換を使用するように **helloworld-mdb** クイックスタートを変更する方法の例になります。

1. **HelloWorldQueueMDB** クラスの **@ActivationConfigProperty destination** プロパティー値を変更して、システムプロパティーの置換を使用します。**@MessageDriven** アノテーションは以下のようになります。

**HelloWorldQueueMDB コードの例**

```
@MessageDriven(name = "HelloWorldQueueMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue =
"${property.helloworldmdb.queue}"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-
acknowledge") })
```

2. **HelloWorldTopicMDB** クラスの **@ActivationConfigProperty destination** プロパティー値を変更して、システムプロパティーの置換を使用します。**@MessageDriven** アノテーションは以下のようになります。

**HelloWorldTopicMDB コードの例**

```
@MessageDriven(name = "HelloWorldQTopicMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue =
"${property.helloworldmdb.topic}"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Topic"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-
acknowledge") })
```

3. **HelloWorldMDBServletClient** クラスの **@Resource** アノテーションを変更して、システムプロパティーの置換を使用します。コードは以下のようになります。

**HelloWorldMDBServletClient コードの例**

```
/**
 * Definition of the two Jakarta Messaging Service destinations used by the quickstart
 * (one queue and one topic).
 */
@JMSDestinationDefinitions(
    value = {
        @JMSDestinationDefinition(
            name = "java:/${property.helloworldmdb.queue}",
            interfaceName = "javax.jms.Queue",
            destinationName = "HelloWorldMDBQueue"
        ),
        @JMSDestinationDefinition(
            name = "java:/${property.helloworldmdb.topic}",
            interfaceName = "javax.jms.Topic",
            destinationName = "HelloWorldMDBTopic"
        )
    })
/**
 * <p>
 * A simple servlet 3 as client that sends several messages to a queue or a topic.
 * </p>
 *
 * <p>
 * The servlet is registered and mapped to /HelloWorldMDBServletClient using the
{@linkplain WebServlet
 * @HttpServlet}.
 * </p>
 *
```

```
 * @author Serge Pagop (spagop@redhat.com)
 *
 */
@WebServlet("/HelloWorldMDBServletClient")
public class HelloWorldMDBServletClient extends HttpServlet {

    private static final long serialVersionUID = -8314035702649252239L;

    private static final int MSG_COUNT = 5;

    @Inject
    private JMSContext context;

    @Resource(lookup = "${property.helloworldmdb.queue}")
    private Queue queue;

    @Resource(lookup = "${property.helloworldmdb.topic}")
    private Topic topic;

  <!-- Remainder of code can be found in the `helloworld-mdb-propertysubstitution` quickstart.
-->
```

4. システムプロパティーの置換値を使用するように **activemq-jms.xml** ファイルを変更します。

**.activemq-jms.xml ファイルの例**

```
<?xml version="1.0" encoding="UTF-8"?>
<messaging-deployment xmlns="urn:jboss:messaging-activemq-deployment:1.0">
    <server>
        <jms-destinations>
            <jms-queue name="HELLOWORLDMDBQueue">
                <entry name="${property.helloworldmdb.queue}"/>
            </jms-queue>
            <jms-topic name="HELLOWORLDMDBTopic">
                <entry name="${property.helloworldmdb.topic}"/>
            </jms-topic>
        </jms-destinations>
    </server>
</messaging-deployment>
```

5. アプリケーションをデプロイします。アプリケーションは、**@Resource** と **@ActivationConfigProperty** プロパティー値のシステムプロパティーで指定された値を使用するようになりました。

## 4.7. アクティベーション設定のプロパティー

### 4.7.1. アノテーションを使用した MDB の設定

アクティベーションプロパティーは、**@ActivationConfigProperty** アノテーションに一致する **@MessageDriven** 要素とサブ要素を使用してアクティベーションプロパティーを設定できます。**@ActivationConfigProperty** は、MDB の設定逢プロパティーのアレイです。**@ActivationConfigProperty** アノテーションの仕様は以下のようになります。

```
@Target(value={})
```

```
@Retention(value=RUNTIME)
public @interface ActivationConfigProperty
{
 String propertyName();
 String propertyValue();
}
```

**@ActivationConfigProperty を表示する例**

```
@MessageDriven(name="MyMDBName",
activationConfig =
{
   @ActivationConfigProperty(propertyName="destinationLookup",propertyValue="queueA"),
   @ActivationConfigProperty(propertyName = "destinationType",propertyValue =
"javax.jms.Queue"),
   @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-
acknowledge"),
})
```

### 4.7.2. デプロイメント記述子を使用した MDB の設定

**ejb-jar.xml** の **<message-driven>** 要素は、Bean を MDB として定義します。**<activation-config>** および要素には、**activation-config-property** 要素を使用した MDB 設定が含まれます。

**サンプル ejb-jar.xml**

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.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-ejb3-2_0.xsd http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
               version="3.1">
   <enterprise-beans>
      <message-driven>
         <ejb-name>MyMDBName</ejb-name>
         <ejb-class>org.jboss.tutorial.mdb_deployment_descriptor.bean.MyMDBName</ejb-class>
         <activation-config>
            <activation-config-property>
               <activation-config-property-name>destinationLookup</activation-config-property-name>
               <activation-config-property-value>queueA</activation-config-property-value>
            </activation-config-property>
            <activation-config-property>
               <activation-config-property-name>destinationType</activation-config-property-name>
               <activation-config-property-value>javax.jms.Queue</activation-config-property-value>
            </activation-config-property>
            <activation-config-property>
               <activation-config-property-name>acknowledgeMode</activation-config-property-name>
               <activation-config-property-value>Auto-acknowledge</activation-config-property-value>
            </activation-config-property>
         </activation-config>
```

```
        </message-driven>
    <enterprise-beans>
</jboss:ejb-jar>
```

表4.1 Jakarta Messaging Service 仕様によって定義されたアクティベーション設定プロパティー

| 名前 | 説明 |
| --- | --- |
| **destinationLookup** | キューまたはトピックの Java Naming and Directory Interface 名。これは必須の値です。 |
| **connectionFactoryLookup** | 管理的に定義された **javax.jms.ConnectionFactory**、**javax.jms.QueueConnectionFactory** または **javax.jms.TopicConnectionFactory** オブジェクトのルックアップ名。これは、エンドポイントがメッセージを受信する Jakarta Messaging プロバイダーへの接続に使用されます。<br><br>明示的に定義されていない場合は、**activemq-ra** という名前のプールされた接続ファクトリーが使用されます。 |
| **destinationType** | 宛先の有効な値のタイプは **javax.jms.Queue** または **javax.jms.Topic** です。これは必須の値です。 |
| **messageSelector** | **messageSelector** プロパティーの値は、利用可能なメッセージのサブセットを選択するために使用される文字列です。この構文は SQL 92 条件式構文のサブセットをベースとし、詳細は Jakarta Messaging 仕様で説明されています。**ActivationSpec** JavaBean 上での **messageSelector** プロパティーの値の指定はオプションです。 |
| **acknowledgeMode** | トランザクション処理された Jakarta Messaging を使用していない場合の承認のタイプ。有効な値は **Auto-acknowledge** または **Dups-ok-acknowledge** です。これは必須の値ではありません。<br><br>デフォルト値は **Auto-acknowledge** です。 |
| **clientID** | 接続のクライアント ID。これは必須の値ではありません。 |
| **subscriptionDurability** | トピックサブスクリプションが永続化されるかどうか。有効な値は **Durable** または **NonDurable** です。これは必須の値ではありません。<br><br>デフォルト値は **NonDurable** です。 |
| **subscriptionName** | トピックサブスクリプションのサブスクリプション名。これは必須の値ではありません。 |

表4.2 JBoss EAP によって定義されるアクティベーション設定プロパティー

| 名前 | 説明 |
| --- | --- |

| 名前 | 説明 |
| --- | --- |
| **destination** | **useJNDI=true** でこのプロパティーを使用すると、**destinationLookup** と同じ意味を持ちます。**useJNDI=false** を指定して使用すると、宛先は検索されませんが、インスタンス化されます。このプロパティーは、**destinationLookup** の代わりに使用できます。これは必須の値ではありません。 |
| **shareSubscriptions** | 接続がサブスクリプションを共有するように設定されているかどうか。<br><br>デフォルト値は **false** です。 |
| **user** | Jakarta Messaging 接続のユーザー。これは必須の値ではありません。 |
| **password** | Jakarta Messaging 接続のパスワード。これは必須の値ではありません。 |
| **maxSession** | 使用する同時セッションの最大数。これは必須の値ではありません。<br><br>デフォルト値は **15** です。 |
| **transactionTimeout** | セッションのトランザクションタイムアウト (ミリ秒単位)。これは必須の値ではありません。<br><br>指定されない場合または 0 の場合、プロパティーは無視され、**transactionTimeout** は上書きされず、Transaction Manager で定義されたデフォルトの **transactionTimeout** が使用されます。 |
| **useJNDI** | Java Naming and Directory Interface を使用して宛先を検索するかどうか。<br><br>デフォルト値は **True** です。 |
| **jndiParams** | 接続で使用する Java Naming および Directory Interface パラメーター。パラメーターは、**;** で区切った **name=value** として定義されます。 |
| **localTx** | XA の代わりにローカルトランザクションを使用します。<br><br>デフォルト値は **false** です。 |
| **setupAttempts** | Jakarta Messaging 接続の設定を試みる回数。Jakarta Messaging リソースが利用可能になる前に MDB がデプロイされる可能性があります。この場合、リソースアダプターは、リソースが利用可能になるまで、複数回の設定を試行します。これは受信接続のみに適用されます。<br><br>デフォルト値は **-1** です。 |
| **setupInterval** | Jakarta Messaging 接続の設定を試みる連続的間隔 (ミリ秒単位)。これは受信接続のみに適用されます。<br><br>デフォルト値は **2000** です。 |

| 名前 | 説明 |
| --- | --- |
| **rebalanceConnections** | 受信接続のリバランスが有効になっているかどうか。このパラメーターは、基礎となるクラスタートポロジーの変更時に受信接続をすべてリバランスできるようにします。<br><br>アウトバウンド接続にはリバランスがありません。<br><br>デフォルト値は **false** です。 |
| **deserializationWhiteList** | ホワイトリストのエントリーのコンマ区切りリスト。これは、信頼できるクラスおよびパッケージのリストです。このプロパティーは、リストにあるオブジェクトをデシリアライズできるように Jakarta Messaging リソースアダプターによって使用されます。<br><br>詳細は、JBoss EAP **Configuring Messaging** の JMS ObjectMessage デシリアライズの制御 を参照してください。 |
| **deserializationBlackList** | ブラックリストのエントリーのコンマ区切りリスト。これは、信頼できないクラスおよびパッケージのリストです。このプロパティーは、リストにあるオブジェクトがデシリアライズされないように、JMS リソースアダプターによって使用されます。<br><br>詳細は、JBoss EAP **Configuring Messaging** の JMS ObjectMessage デシリアライズの制御 を参照してください。 |

### 4.7.3. MDB を設定するためのいくつかのユースケース

- メッセージを受信する MDB のユースケース
  MDB がメッセージを受信する際の基本的なシナリオは、JBoss EAP に同梱される **helloworld-mdb** クイックスタートを参照してください。

- メッセージを送信する MDB のユースケース
  メッセージの処理後、他のビジネスシステムに通知したり、メッセージに応答する必要がある場合があります。この場合、以下のスニペットにあるように、MDB からメッセージを送信できます。

```
package org.jboss.as.quickstarts.mdb;

import javax.annotation.Resource;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.inject.Inject;
import javax.jms.JMSContext;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;

@MessageDriven(name = "MyMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue =
"queue/MyMDBRequest"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
```

```
"javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-
acknowledge") })
public class MyMDB implements MessageListener {

    @Inject
    private JMSContext jmsContext;

    @Resource(lookup = "java:/queue/ResponseDefault")
    private Queue defaultDestination;

    /**
     * @see MessageListener#onMessage(Message)
     */
    public void onMessage(Message rcvMessage) {
        try {
            Message response = jmsContext.createTextMessage("Response for message " +
rcvMessage.getJMSMessageID());
            if (rcvMessage.getJMSReplyTo() != null) {
                jmsContext.createProducer().send(rcvMessage.getJMSReplyTo(), response);
            } else {
                jmsContext.createProducer().send(defaultDestination, response);
            }
        } catch (JMSException e) {
            throw new RuntimeException(e);
        }
    }
}
```

上記の例では、MDB はメッセージを受信した後、**JMSReplyTo** で指定された宛先、または Java Naming and Directory Interface 名 **java:/queue/ResponseDefault** にバインドされた宛先のいずれかに応答します。

- MDB でのユースケースインバウンド接続のリバランスの設定

```
@MessageDriven(name="MyMDBName",
    activationConfig =
    {
        @ActivationConfigProperty(propertyName = "destinationType",propertyValue =
"javax.jms.Queue"),
        @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue =
"queueA"),
        @ActivationConfigProperty(propertyName = "rebalanceConnections", propertyValue =
"true")
    }
)
```

# 第5章 セッション BEAN の呼び出し

## 5.1. EJB クライアントコンテキスト

JBoss EAP には、リモート EJB 呼び出しを管理するための EJB クライアント API が導入されました。JBoss EJB クライアント API は、EJBClientContext を使用します。これは、1 つ以上のスレッドで割り当てられ、複数のスレッドと同時に使用される可能性があります。つまり、EJBClientContext には任意の数の EJB レシーバーが含まれる可能性があります。EJB レシーバーは、EJB 呼び出しを処理できるサーバーと通信する方法を認識しているコンポーネントです。通常、EJB リモートアプリケーションは以下のように分類できます。

- スタンドアロン Java アプリケーションとして実行されるリモートクライアント。
- 別の JBoss EAP インスタンス内で実行されるリモートクライアント。

リモートクライアントのタイプによっては、EJB クライアント API の観点からすべば、JVM 内に複数の EJBClientContext が存在する可能性があります。

通常、スタンドアロンアプリケーションには任意の数の EJB レシーバーでサポートされる単一の EJBClientContext がありますが、これは必須ではありません。スタンドアロンアプリケーションに複数の EJBClientContext がある場合、EJB クライアントコンテキストセレクターは適切なコンテキストを返します。

別の JBoss EAP インスタンス内で実行されるリモートクライアントの場合には、デプロイされた各アプリケーションに対応する EJB クライアントコンテキストがあります。アプリケーションが別の EJB を呼び出すたびに、対応する EJB クライアントコンテキストを使用して正しい EJB レシーバーを検索し、呼び出しを処理します。

## 5.2. リモート EJB クライアントの使用

### 5.2.1. 初期コンテキストルックアップ

初期コンテキストの作成時に **PROVIDER_URL** プロパティーを使用してリモートサーバーのアドレスを渡すことができます。

```
public class Client {
    public static void main(String[] args)
            throws NamingException, PrivilegedActionException, InterruptedException {
        InitialContext ctx = new InitialContext(getCtxProperties());
        String lookupName = "ejb:/server/HelloBean!ejb.HelloBeanRemote";
        HelloBeanRemote bean = (HelloBeanRemote)ctx.lookup(lookupName);
        System.out.println(bean.hello());
        ctx.close();
    }
    public static Properties getCtxProperties() {
        Properties props = new Properties();
        props.put(Context.INITIAL_CONTEXT_FACTORY,
WildFlyInitialContextFactory.class.getName());
        props.put(Context.PROVIDER_URL, "remote+http://127.0.0.1:8080");
        props.put(Context.SECURITY_PRINCIPAL, "joe");
        props.put(Context.SECURITY_CREDENTIALS, "joeIsAwesome2013!");
        return props;
    }
}
```

**注記**

ルックアップに使用される初期コンテキストファクトリーは **org.wildfly.naming.client.WildFlyInitialContextFactory** です。

### 5.2.2. リモート EJB 設定ファイル

JBoss EAP は Elytron セキュリティーフレームワークを特長としています。クライアントアプリケーションのクラスパスの **META-INF/** ディレクトリーにある **wildfly-config.xml** ファイルは、Elytron セキュリティーフレームワークおよび EJB クライアント設定に幅広い認証および承認オプションを許可します。

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <authentication-rules>
      <rule use-configuration="default" />
    </authentication-rules>
    <authentication-configurations>
      <configuration name="default">
        <sasl-mechanism-selector selector="DIGEST-MD5" />
        <set-user-name name="admin" />
        <credentials>
          <clear-password password="password123!" />
        </credentials>
      </configuration>
    </authentication-configurations>
  </authentication-client>
  <jboss-ejb-client xmlns="urn:jboss:wildfly-client-ejb:3.0">
    <connections>
      <connection uri="remote+http://127.0.0.1:8080" />
    </connections>
  </jboss-ejb-client>
</configuration>
```

初期コンテキストに **PROVIDER_URL**、**SECURITY_PRINCIPAL**、および **SECURITY_CREDENTIALS** パラメーターを組み込む代わりに、**wildfly-config.xml** ファイルの **<connection-uri>** と **<authentication-client>** 要素を使用して接続 URI とセキュリティーを設定できます。

### 5.2.3. ClientTransaction アノテーション

**@org.jboss.ejb.client.annotation.ClientTransaction** アノテーションは、EJB クライアントからのトランザクション伝搬を処理します。クライアントにトランザクションがない場合には伝搬に失敗し、クライアントがアクティブであってもトランザクションの伝搬を防ぐことができます。**org.jboss.ejb.client.annotation.ClientTransactionPolicy** インターフェイスの定数を使用して、**ClientTransaction** アノテーションのポリシーを制御できます。以下は、**org.jboss.ejb.client.annotation.ClientTransactionPolicy** インターフェイスの定数です。

- MANDATORY: クライアント側のトランザクションコンテキストがない場合は例外で失敗します。クライアント側のトランザクションコンテキストが存在する場合は伝播します。
- NEVER: トランザクションコンテキストを伝搬することなく起動。クライアント側のトランザクションコンテキストが存在する場合は、例外が発生します。
- NOT_SUPPORTED: クライアント側のトランザクションコンテキストが存在するかどうかにかかわらず、トランザクションコンテキストを伝搬せずに起動します。

- SUPPORTS: クライアント側のトランザクションコンテキストがない場合にトランザクションなしで起動します。クライアント側のトランザクションコンテキストが存在する場合は伝播します。

アノテーションがない場合、デフォルトのポリシーは **org.jboss.ejb.client.annotation.ClientTransactionPolicy#SUPPORTS** になります。存在する場合は、トランザクションのありなしに関係なく、伝搬は失敗しません。

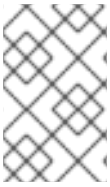
```
@ClientTransaction(ClientTransactionPolicy.MANDATORY)
@Remote
public interface RemoteCalculator  {
   public void callRemoteEjb() { }
}
@Stateless
@Remote(RemoteCalculator.class)
public class CalculatorBean implements RemoteCalculator {

   @Override
   public void callRemoteEjb()  {   }
}
```

アノテーションでは、リモートインターフェイスプロバイダーがメソッドにトランザクションが必要であるかどうかをリモートインターフェイスのコンシューマーに指示できるようにします。

## 5.3. リモート EJB データ圧縮

これまでのバージョンの JBoss EAP には、EJB プロトコルメッセージが含まれるメッセージストリームを圧縮できる機能が含まれていました。この機能は JBoss EAP 6.3 以降に含まれています。



**注記**

圧縮は現在、クライアントおよびサーバー側にあるべき EJB インターフェイス上のアノテーションでのみ指定できます。現在、圧縮ヒントを指定するのと同等の XML はありません。

データ圧縮ヒントは、JBoss アノテーション **org.jboss.ejb.client.annotation.CompressionHint** で指定できます。ヒントの値では、リクエスト、応答、要求、応答を圧縮するかどうかを指定します。**@CompressionHint** デフォルトを **compressResponse=true** と **compressRequest=true** に追加。

アノテーションはインターフェイスレベルで指定して、以下のように EJB のインターフェイス内のすべてのメソッドに適用できます。

```
import org.jboss.ejb.client.annotation.CompressionHint;

@CompressionHint(compressResponse = false)
public interface ClassLevelRequestCompressionRemoteView {
    String echo(String msg);
}
```

また、アノテーションは EJB のインターフェイスにある特定のメソッドに適用できます。以下に例を示します。

```
import org.jboss.ejb.client.annotation.CompressionHint;
```

```
public interface CompressableDataRemoteView {

    @CompressionHint(compressResponse = false, compressionLevel =
Deflater.BEST_COMPRESSION)
    String echoWithRequestCompress(String msg);

    @CompressionHint(compressRequest = false)
    String echoWithResponseCompress(String msg);

    @CompressionHint
    String echoWithRequestAndResponseCompress(String msg);

    String echoWithNoCompress(String msg);
}
```

上記の **compressionLevel** 設定は、以下の値にすることができます。

- BEST_COMPRESSION
- BEST_SPEED
- DEFAULT_COMPRESSION
- NO_COMPRESSION

**CompressionLevel** のデフォルト設定は **Deflater.DEFAULT_COMPRESSION** です。

メソッドレベルでのクラスレベルのアノテーションの上書き:

```
@CompressionHint
public interface MethodOverrideDataCompressionRemoteView {

    @CompressionHint(compressRequest = false)
    String echoWithResponseCompress(final String msg);

    @CompressionHint(compressResponse = false)
    String echoWithRequestCompress(final String msg);

    String echoWithNoExplicitDataCompressionHintOnMethod(String msg);
}
```

クライアント側で、**org.jboss.ejb.client.view.annotation.scan.enabled** システムプロパティーが **true** に設定されていることを確認します。このプロパティーは JBoss EJB Client に対してアノテーションをスキャンするように指示します。

## 5.4. EJB クライアントのリモーティング相互運用性

デフォルトのリモート接続ポートは **8080** です。**Jboss-ejb-client** プロパティーファイルは以下のようになります。

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
remote.connection.default.host=localhost
```

```
remote.connection.default.port=8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

**デフォルトのコネクター**
デフォルトのコネクターは **http-remoting** です。

- クライアントアプリケーションが JBoss EAP 6 の EJB クライアントライブラリーを使用し、JBoss EAP 7 サーバーへ接続する場合、**8080** 以外のポートでリモーティングコネクターを公開するようサーバーを設定する必要があります。その後、そのクライアントは新しく設定されたコネクターを使用して接続する必要があります。

- JBoss EAP 7 の EJB クライアントライブラリーを使用し、JBoss EAP 6 サーバーへ接続するクライアントアプリケーションは、サーバーインスタンスによって **http-remoting** コネクターは使用されず、**remoting** コネクターが使用されることを認識する必要があります。これは、新しいクライアント側接続プロパティーを定義することで実現されます。

  ```
  remote.connection.default.protocol=remote
  ```

**注記**

EJB リモート呼び出しは、JBoss EAP 7 と JBoss EAP 6 のみでサポートされます。

EJB クライアントのリモーティング相互運用性のほかにも、以下のオプションを使用してレガシークライアントに接続できます。

- JBoss EAP **設定ガイド**の JTS トランザクションの ORB の設定 します。

## 5.5. リモート EJB 呼び出しの IIOP の設定

JBoss EAP は、JBoss EAP にデプロイされた EJB への CORBA/IIOP ベースのアクセスをサポートします。

**<iiop>** 要素は、IIOP、CORBA、EJB の呼び出しを有効にするために使用されます。この要素が存在するということは、**iiop-openjdk** サブシステムがインストールされていることを意味します。**<iiop>** 要素には、以下の属性が含まれます。

- **enable-by-default**: これが **true** の場合、EJB 2.x のホームインターフェイスを持つすべての EJB は IIOP 経由で公開されます。それ以外の場合は、**jboss-ejb3.xml** を使用して明示的に有効にする必要があります。

- **use-qualified-name**: これが **true** の場合、EJB は **myear/myejbjar/MyBean** などのデプロイメントのアプリケーションおよびモジュール名が含まれるバインディング名で CORBA ネーミングコンテキストにバインドされます。**false**、デフォルトのバインディング名は bean 名のみになります。

**注記**

通常、**RemoteHome** インターフェイスは EJB 3 リモート呼び出しには必要ありませんが、IIOP を使用して公開される EJB 3 Bean に必要です。次に、**jboss-ejb3.xml** ファイルを使用して IIOP を有効にするか、**standalone-full.xml** 設定ファイル内のすべての EJB に対して IIOP を有効にする必要があります。

**IIOP の有効化**

IIOP を有効にするには、IIOP OpenJDK ORB サブシステムがインストールされており、**ejb3** サブシステム設定にある **<iiop/>** 要素がインストールされている必要があります。ディストリビューションに同梱される **standalone-full.xml** 設定は、どちらも有効になっています。

IIOP は、サーバー設定ファイルの **iiop-openjdk** サブシステムで設定されます。

```
<subsystem xmlns="urn:jboss:domain:iiop-openjdk:2.1">
```

以下の管理 CLI コマンドを使用して **iiop-openjdk** サブシステムにアクセスし、更新します。

```
/subsystem=iiop-openjdk
```

IIOP 要素は、サーバーのデフォルト動作を制御する 2 つの属性を取ります。

```
<subsystem xmlns="urn:jboss:domain:ejb3:5.0">
  ...
  <iiop enable-by-default="false" use-qualified-name="false"/>
  ...
</subsystem>
```

以下の管理 CLI コマンドは、**ejb3** サブシステムの下に **<iiop>** 要素を追加します。

```
/subsystem=ejb3/service=iiop:add(enable-by-default=false, use-qualified-name=false)
```

### IIOP を使用して通信する EJB の作成

以下の例は、クライアントからリモート IIOP 呼び出しを作成する方法を示しています。

1. サーバー上で EJB 2 Bean を作成します。

```
@Remote(IIOPRemote.class)
@RemoteHome(IIOPBeanHome.class)
@Stateless
public class IIOPBean {
    public String sayHello() throws RemoteException {
        return "hello";
    }
}
```

2. 必須メソッド **create()** を持つホーム実装を作成します。この方法は、ビジネスメソッドを呼び出すリモートインターフェイスのプロキシーを取得するためにクライアントによって呼び出されます。

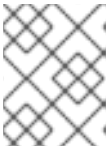
```
public interface IIOPBeanHome extends EJBHome {
    public IIOPRemote create() throws RemoteException;
}
```

3. EJB へのリモート接続用のリモートインターフェイスを作成します。

```
public interface IIOPRemote extends EJBObject {
    String sayHello() throws RemoteException;
}
```

4. **META-INF** に記述子ファイル **jboss-ejb3.xml** を作成して、リモート呼び出しの Bean を導入します。

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:iiop="urn:iiop"
               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-ejb3-2_0.xsd
                 http://java.sun.com/xml/ns/javaee http://www.jboss.org/j2ee/schema/jboss-ejb3-
spec-2_0.xsd
                 urn:iiop jboss-ejb-iiop_1_0.xsd"
               version="3.1"
               impl-version="2.0">
   <assembly-descriptor>
      <iiop:iiop>
         <ejb-name>*</ejb-name>
      </iiop:iiop>
   </assembly-descriptor>
</jboss:ejb-jar>
```



**注記**

JAR ファイルの記述子とともに、パッケージ化した Bean を JBoss EAP コンテナーにデプロイする準備が整いました。

5. クライアント側でコンテキストを作成します。

```
System.setProperty("com.sun.CORBA.ORBUseDynamicStub", "true");
final Properties props = new Properties();
props.put(Context.PROVIDER_URL, "corbaloc::localhost:3528/JBoss/Naming/root");
props.setProperty(Context.URL_PKG_PREFIXES,
"org.jboss.iiop.naming:org.jboss.naming.client");
props.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.cosnaming.CNCtxFactory");
props.put(Context.OBJECT_FACTORIES,
"org.jboss.tm.iiop.client.IIOPClientUserTransactionObjectFactory");
```

**注記**

クライアントでは、**wildfly iiop openjdk** ライブラリーをクラスパスに追加する必要があります。また、クライアントは **org.wildfly:wildfly-iiop-openjdk** アーティファクトを Maven 依存関係として追加する必要がある場合があります。

6. コンテキストルックアップを使用して **IIOPBeanHome** ホームディレクトリーへの参照を絞り込みます。次に、ホームインターフェイス **create()** メソッドを呼び出し、リモートインターフェイスにアクセスします。これにより、そのメソッドを呼び出すことができます。

```
try {
   Context context = new InitialContext(props);

   final Object iiopObj = context.lookup(IIOPBean.class.getSimpleName());
   final IIOPBeanHome beanHome = (IIOPBeanHome)
```

```
PortableRemoteObject.narrow(iiopObj, IIOPBeanHome.class);
    final IIOPRemote bean = beanHome.create();

    System.out.println("Bean saying: " + bean.sayHello());
} catch (Exception e) {
    e.printStackTrace();
}
```

## 5.6. EJB クライアントアドレスの設定

以下の例のように、**SessionContext** インターフェイスを使用して EJB クライアントアドレスを確認できます。

```
public class HelloBean implements HelloBeanRemote {
    @Resource
    SessionContext ctx;
    private Long counter;
    public HelloBean() {
    }
    @PostConstruct
    public void init() {
        counter = 0L;
    }
    @Override
    @RolesAllowed("users")
    public String hello() {
        final String message = "method hello() invoked by user " + ctx.getCallerPrincipal().getName()
                + ", source addr = " +  ctx.getContextData().get("jboss.source-address").toString();
        System.out.println(message);
        return message;
    }
}
```

**スタンドアロンクライアント設定**
**wildfly-config.xml** ファイルに名前空間 **urn:xnio:3.5** がある **worker** 要素内に **outbound-bind-addresses** 要素を設定できます。**bind-address** サブ要素は、以下のように定義される属性 **match**、**bind-address**、**bind-port** を取ります。

以下は、**wildfly-config.xml** ファイルを使用したスタンドアロンクライアント設定の例です。

```
<configuration>
    <worker xmlns="urn:xnio:3.5">
        <worker-name value="default"/>
        <outbound-bind-addresses>
            <bind-address bind-address=IP_ADDRESS_TO_BIND_TO bind-
port=OPTIONAL_SOURCE_PORT_NUMBER match=CIDR_BLOCK />
        </outbound-bind-addresses>
    </worker>
</configuration>
```

**Outbound-bind-address** には以下の属性が必要です。

- **match** は、**10.0.0.0/8**、**ff00::\8**、**0.0.0.0/0**、**::/0** などの Classless Inter-Domain Routing (CIDR) ブロックです。

- **bind-address** は、宛先アドレスが **match** パラメーターで指定された CIDR ブロックと一致する場合にバインドする IP アドレスを指定します。CIDR ブロックと同じアドレスファミリーである必要があります。

- **bind-port** は任意のソースポート番号で、デフォルトは **0** です。
一致する式が存在しない場合は、アウトバウンドソケットが明示的にバインドされません。

**コンテナーベースの設定**
EJB クライアントアドレスのコンテナーベースの設定は、**wildfly-config.xml** ファイルで定義されたスタンドアロンクライアント設定と似ています。

以下の例は、**ejb3** サブシステムがデフォルトで使用する **io** サブシステムのデフォルトの **worker** 要素で **outbound-bind-address** を設定します。

```
/subsystem=io/worker=default/outbound-bind-address=SPECIFY_OUTBOUND_BIND_ADDRESS:add(bind-address=IP_ADDRESS_TO_BIND_TO, bind-port=OPTIONAL_SOURCE_PORT_NUMBER, match=CIDR_BLOCK)
```

## 5.7. HTTP 上の EJB 呼び出し

HTTP 上の EJB 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。

### 5.7.1. クライアント側実装

クライアント側の実装は、Undertow HTTP クライアントを使用してサーバーを呼び出す **EJBReceiver** で設定されます。接続管理は、接続プールを使用して自動的に処理されます。

HTTP トランスポートを使用するようにリモート EJB 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-ejb-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。

### 5.7.2. サーバー側実装

サーバー側実装は、受信 HTTP リクエストを処理し、それらをアンマーシャルし、結果を内部 EJB 呼び出しコードに渡すサービスで設定されます。

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で **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できます。

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

**http-invoker** には **wildfly-services** がデフォルトの **path** と、以下のいずれかの属性があります。

- 上記のコマンドが示すように、Elytron **http-authentication-factory** への参照である必要がある **http-authentication-factory**。
- レガシー **security-realm**.

上記の属性は相互排他的であることに注意してください。**http-authentication-factory** と **security-realm** の両方を同時に指定することはできません。

**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 セキュリティーと、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ードメインを使用する必要があります。

# 第6章 EJB アプリケーションセキュリティー

## 6.1. セキュリティーアイデンティティー

### 6.1.1. EJB セキュリティーアイデンティティー

EJB は、他のコンポーネントでメソッドを呼び出す際に使用するアイデンティティーを指定できます。これは、EJB セキュリティーアイデンティティーで、呼び出しアイデンティティーとも呼ばれます。

デフォルトでは、EJB は独自の呼び出し元アイデンティティーを使用します。アイデンティティーは、特定のセキュリティーロールに設定することもできます。特定のセキュリティーロールの使用は、セグメント化されたセキュリティーモデルを構築する場合に便利です。たとえば、内部 EJB のみへのコンポーネントセットへのアクセスを制限します。

### 6.1.2. EJB のセキュリティーアイデンティティーの設定

EJB のセキュリティーアイデンティティーは、セキュリティー設定の **<security-identity>** タグで指定します。**<security-identity>** タグが存在する場合は、EJB の呼び出し元アイデンティティーがデフォルトで使用されます。

**例: Enterprise JavaBeans のセキュリティーアイデンティティーを同じ呼び出し元として設定**

この例では、EJB によって作成されたメソッド呼び出しのセキュリティーアイデンティティーを、現在の呼び出し元のアイデンティティーと同じに設定します。**<security-identity>** 要素の宣言を指定しない場合は、この動作がデフォルトになります。

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ASessionBean</ejb-name>
      ...
      <security-identity>
        <use-caller-identity/>
      </security-identity>
    </session>
    ...
  </enterprise-beans>
</ejb-jar>
```

**例: Enterprise JavaBeans のセキュリティーアイデンティティーの特定のロールへの設定**

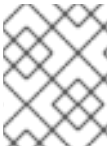
セキュリティーアイデンティティーを特定のロールに設定するには、**<security-identity>** タグ内の **<run-as>** と **<role-name>** タグを使用します。

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>RunAsBean</ejb-name>
      ...
      <security-identity>
        <run-as>
          <description>A private internal role</description>
          <role-name>InternalRole</role-name>
        </run-as>
```

```
      </security-identity>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>
```

デフォルトでは、**<run-as>** を使用すると、**anonymous** という名前のプリンシパルが発信呼び出しに割り当てられます。異なるプリンシパルを割り当てるには、**<run-as-principal>** を使用します。

```
<session>
   <ejb-name>RunAsBean</ejb-name>
   <security-identity>
      <run-as-principal>internal</run-as-principal>
   </security-identity>
</session>
```



**注記**

また、servlet 要素内に **<run-as>** および **<run-as-principal>** 要素を使用することもできます。

## 6.2. EJB メソッドパーミッション

### 6.2.1. EJB メソッドパーミッションについて

EJB はメソッドへのアクセスを特定のセキュリティーロールに制限できます。

EJB **<method-permission>** 要素の宣言は、EJB のインターフェイスメソッドを呼び出すことができるロールを指定します。以下の組み合わせのパーミッションを指定できます。

- 名前付き EJB のすべてのホームおよびコンポーネントインターフェイスメソッド
- 名前付き EJB のホームまたはコンポーネントインターフェイスの指定メソッド
- オーバーロード名を持つ一連のメソッドに指定されたメソッド

### 6.2.2. EJB メソッドパーミッションの使用

**<method-permission>** 要素は、**<method>** 要素で定義される EJB メソッドへのアクセスを許可される論理ロールを定義します。複数の例は、xml の構文を示しています。複数のメソッドパーミッションステートメントが存在する可能性があり、それらのステートメントには累積的影響があります。**<method-permission>** 要素は、**<ejb-jar>** 記述子の **<assembly-descriptor>** 要素の子です。

XML 構文は、EJB メソッドパーミッションのアノテーションを使用する代わりになります。

**例: ロールによるエンタープライズ JavaBean のすべてのメソッドへのアクセスを許可する**

```
<method-permission>
  <description>The employee and temp-employee roles may 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>
```

**例: Enterprise JavaBeans および制限メソッドパラメーターの特定のメソッドにロールがアクセスできるようにする**

```
<method-permission>
  <description>The employee role may access the findByPrimaryKey,
  getEmployeeInfo, and the updateEmployeeInfo(String) method of
  the AcmePayroll bean </description>
  <role-name>employee</role-name>
  <method>
    <ejb-name>AcmePayroll</ejb-name>
    <method-name>findByPrimaryKey</method-name>
  </method>
  <method>
    <ejb-name>AcmePayroll</ejb-name>
    <method-name>getEmployeeInfo</method-name>
  </method>
  <method>
    <ejb-name>AcmePayroll</ejb-name>
    <method-name>updateEmployeeInfo</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </method>
</method-permission>
```

**例: 任意の認証されたユーザーによる Enterprise JavaBeans のメソッドへのアクセスの許可**

**<unchecked/>** 要素を使用すると、認証されたユーザーが指定されたメソッドを使用できます。

```
<method-permission>
  <description>Any authenticated user may access any method of the
  EmployeeServiceHelp bean</description>
  <unchecked/>
  <method>
    <ejb-name>EmployeeServiceHelp</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

**例: 特定のエンタープライズ JavaBeans メソッドを完全に除外**

```
<exclude-list>
  <description>No fireTheCTO methods of the EmployeeFiring bean may be
  used in this deployment</description>
  <method>
    <ejb-name>EmployeeFiring</ejb-name>
    <method-name>fireTheCTO</method-name>
  </method>
</exclude-list>
```

**例: 複数の <method-permission> ブロックを含む完全な <assembly-descriptor>**

```
<ejb-jar>
   <assembly-descriptor>
      <method-permission>
         <description>The employee and temp-employee roles may 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>
      <method-permission>
         <description>The employee role may access the findByPrimaryKey, getEmployeeInfo, and the
updateEmployeeInfo(String) method of the AcmePayroll bean </description>
         <role-name>employee</role-name>
         <method>
            <ejb-name>AcmePayroll</ejb-name>
            <method-name>findByPrimaryKey</method-name>
         </method>
         <method>
            <ejb-name>AcmePayroll</ejb-name>
            <method-name>getEmployeeInfo</method-name>
         </method>
         <method>
            <ejb-name>AcmePayroll</ejb-name>
            <method-name>updateEmployeeInfo</method-name>
            <method-params>
               <method-param>java.lang.String</method-param>
            </method-params>
         </method>
      </method-permission>
      <method-permission>
         <description>The admin role may access any method of the EmployeeServiceAdmin bean
</description>
         <role-name>admin</role-name>
         <method>
            <ejb-name>EmployeeServiceAdmin</ejb-name>
            <method-name>*</method-name>
         </method>
      </method-permission>
      <method-permission>
         <description>Any authenticated user may access any method of the EmployeeServiceHelp
bean</description>
         <unchecked/>
         <method>
            <ejb-name>EmployeeServiceHelp</ejb-name>
            <method-name>*</method-name>
         </method>
      </method-permission>
      <exclude-list>
         <description>No fireTheCTO methods of the EmployeeFiring bean may be used in this
deployment</description>
         <method>
```

```
            <ejb-name>EmployeeFiring</ejb-name>
            <method-name>fireTheCTO</method-name>
          </method>
        </exclude-list>
      </assembly-descriptor>
    </ejb-jar>
```

## 6.3. EJB セキュリティーアノテーション

### 6.3.1. EJB セキュリティーアノテーションについて

EJB **javax.annotation.security** アノテーションは JSR-250 で定義されます。これらのアノテーションの Jakarta と同等の Jakarta は、Jakarta Annotations 1.3 仕様 で定義されています。

EJB はセキュリティーアノテーションを使用してセキュリティーに関する情報をデプロイヤーに渡します。これらの型には次のようなものがあります。

**@DeclareRoles**

利用可能なロールを宣言します。

**@RunAs**

コンポーネントの伝播セキュリティーアイデンティティーを設定します。

### 6.3.2. EJB セキュリティーアノテーションの使用

XML 記述子またはアノテーションのいずれかを使用して、EJB でメソッドを呼び出すことのできるセキュリティーロールを制御できます。XML 記述子の使用方法は、Use EJB Method Permissions を参照してください。

デプロイメント記述子に明示的に指定されるメソッドの値はすべて、アノテーションの値を上書きします。メソッドの値がデプロイメント記述子で指定されていない場合は、アノテーションを使用して設定された値が使用されます。上書きの粒度は、方法ごとに決定されます。

**Enterprise JavaBeans のセキュリティーパーミッションを制御するアノテーション**

**@DeclareRoles**

**@DeclareRoles** を使用して、パーミッションをチェックするセキュリティーロールを定義します。**@DeclareRoles** がない場合、一覧は **@RolesAllowed** アノテーションから自動的にビルドされます。ロールの設定に関する詳細は、Java EE チュートリアル Specifying Authorized Users by Declaring Security Roles を参照してください。

**@RolesAllowed, @PermitAll, @DenyAll**

**@RolesAllowed** を使用して、単一のメソッドまたは複数のメソッドへのアクセスを許可するロールを一覧表示します。単一のメソッドまたは複数のメソッドの使用からすべてのロールを許可または拒否するには、**@PermitAll** または **@DenyAll** を使用します。アノテーションメソッドのパーミッション設定に関する詳細は、Java EE チュートリアルの Specifying Authorized Users by Declaring Security Roles を参照してください。

**@RunAs**

**@RunAs** を使用して、アノテーション付きのメソッドから呼び出しを行うときにメソッドが使用するロールを指定します。アノテーションを使用してセキュリティーアイデンティティーを伝播する方法は、Java EE チュートリアルの Propagating a Security Identity (Run-As) を参照してください。

**例: セキュリティーアノテーションの例**

-

```
@Stateless
@RolesAllowed({"admin"})
@SecurityDomain("other")
public class WelcomeEJB implements Welcome {
    @PermitAll
    public String WelcomeEveryone(String msg) {
        return "Welcome to " + msg;
    }
    @RunAs("tempemployee")
    public String GoodBye(String msg) {
        return "Goodbye, " + msg;
    }
    public String GoodbyeAdmin(String msg) {
        return "See you later, " + msg;
    }
}
```

このコードでは、すべてのロールが **WelcomeEveryone** メソッドにアクセスできます。**GoodBye** メソッドは、呼び出しを行う際に **tempemployee** ロールを使用します。メソッド **GoodbyeAdmin** や、セキュリティーアノテーションの他のメソッドには、**admin** ロールのみがアクセスできます。

## 6.4. EJB へのリモートアクセス

### 6.4.1. リモート EJB クライアントでのセキュリティーレルムの使用

EJB をリモートで呼び出すクライアントにセキュリティーを追加する方法として、セキュリティーレルムを使用することができます。セキュリティーレルムは、ユーザー名/パスワードのペアとユーザー名/ロールのペアの単純なデータベースです。この用語は Web コンテナーのコンテキストでも使用され、意味が若干異なります。

EJB に対してセキュリティーレルムに存在する特定のユーザー名/パスワードのペアを認証するには、以下の手順に従います。

- 新しいセキュリティーレルムをドメインコントローラーまたはスタンドアロンサーバーに追加します。
- 以下の例のように、アプリケーションのクラスパスにある **wildfly-config.xml** ファイルを設定します。

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <authentication-rules>
      <rule use-configuration="default" />
    </authentication-rules>
    <authentication-configurations>
      <configuration name="default">
        <sasl-mechanism-selector selector="DIGEST-MD5" />
        <set-user-name name="admin" />
        <credentials>
          <clear-password password="password123!" />
        </credentials>
      </configuration>
    </authentication-configurations>
  </authentication-client>
  <jboss-ejb-client xmlns="urn:jboss:wildfly-client-ejb:3.0">
```

```
        <connections>
          <connection uri="remote+http://127.0.0.1:8080" />
        </connections>
      </jboss-ejb-client>
    </configuration>
```

- 新しいセキュリティーレルムを使用するドメインまたはスタンドアロンサーバーでカスタムリモーティングコネクターを作成します。
- カスタムリモーティングコネクターでプロファイルを使用するよう設定されたサーバーグループに EJB を追加します。または管理対象ドメインを使用していない場合はスタンドアロンサーバーに EJB をデプロイします。

### 6.4.2. 新しいセキュリティーレルムの追加

1. 管理 CLI の実行:
   **jboss-cli.sh** または **jboss-cli.bat** スクリプトを実行し、サーバーに接続します。
2. 新しいセキュリティーレルム自体を作成します。
   以下のコマンドを実行し、ドメインコントローラーまたはスタンドアロンサーバーに **MyDomainRealm** という名前の新しいセキュリティーレルムを作成します。

   ドメインインスタンスの場合は、以下のコマンドを使用します。

   ```
   /host=master/core-service=management/security-realm=MyDomainRealm:add()
   ```

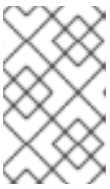
   スタンドアロンインスタンスの場合には、以下のコマンドを使用します。

   ```
   /core-service=management/security-realm=MyDomainRealm:add()
   ```

3. **myfile.properties** という名前のプロパティーファイルを作成します。
   スタンドアロンインスタンスの場合は、**EAP_HOME/standalone/configuration/myfile.properties** ファイルを作成し、ドメインインスタンスの場合は **EAP_HOME/domain/configuration/myfile.properties** ファイルを作成します。これらのファイルには、ファイル所有者の読み取りおよび書き込みアクセスが必要です。

   ```
   $ chmod 600 myfile.properties
   ```

4. 新規ロールについての情報を格納するプロパティーファイルへの参照を作成します。
   以下のコマンドを実行して、新しいロールに関連するプロパティーが含まれる **myfile.properties** ファイルへのポインターを作成します。

   **注記**

   プロパティーファイルは、含まれる **add-user.sh** スクリプトおよび **add-user.bat** スクリプトでは作成されません。これは外部に作成する必要があります。

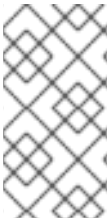
   ドメインインスタンスの場合は、以下のコマンドを使用します。

   ```
   /host=master/core-service=management/security-
   realm=MyDomainRealm/authentication=properties:add(path=myfile.properties)
   ```

スタンドアロンインスタンスの場合には、以下のコマンドを使用します。

```
/core-service=management/security-
realm=MyDomainRealm/authentication=properties:add(path=myfile.properties)
```

新しいセキュリティーレルムが作成されます。この新しいレルムにユーザーおよびロールを追加すると、情報はデフォルトのセキュリティーレルムとは別のファイルに保存されます。この新しいファイルは、独自のアプリケーションまたは手順を使用して管理できます。



**注記**

**add-user.sh** スクリプトを使用して、**application-users.properties** 以外のデフォルト以外のファイルにユーザーを追加する場合は、引数 **--user-properties myfile.properties** を渡す必要があります。それ以外の場合は、**application-users.properties** の使用を試行します。

### 6.4.3. セキュリティーレルムへのユーザーの追加

1. **add-user** スクリプトを実行します。ターミナルを開き、**EAP_HOME/bin/** ディレクトリーに移動します。Red Hat Enterprise Linux またはその他の UNIX と同様のオペレーティングシステムを使用している場合には、**add-user.sh** を実行します。Windows Server を使用している場合は、**add-user.bat** を実行します。

2. 管理ユーザーまたはアプリケーションユーザーを追加するかどうかを選択します。この手順では、**b** と入力してアプリケーションユーザーを追加します。

3. ユーザーを追加するレルムを選択します。デフォルトでは、利用可能なレルムは **ApplicationRealm** のみです。カスタムレルムを追加してしまった場合は、その代わりにユーザーを追加します。

4. プロンプトが表示されたら、ユーザー名、パスワード、およびロールを入力します。プロンプトが表示されたら、希望のユーザー名、パスワード、および任意のロールを入力します。**yes** を入力して選択を確認するか、**no** と入力して変更をキャンセルします。変更は、セキュリティーレルムの各プロパティーファイルに書き込まれます。

### 6.4.4. セキュリティードメインとセキュリティーレルム間の関係

**重要**

EJB をセキュリティーレルムでセキュアにするには、セキュリティーレルムからユーザーの認証情報を取得するように設定したセキュリティードメインを使用する必要があります。これは、ドメインに Remoting および RealmDirect ログインモジュールが必要であることを意味します。セキュリティードメインの割り当ては、EJB で適用可能な **@SecurityDomain** アノテーションによって実行されます。

**other** セキュリティードメインは、基礎となるセキュリティーレルムからユーザーとパスワードデータを取得します。EJB に **@SecurityDomain** アノテーションがなく、EJB にセキュリティーが保護されていると思われる他のセキュリティー関連のアノテーションのいずれかが含まれる場合は、このセキュリティードメインがデフォルトのドメインになります。

接続を確立するためにクライアントによって使用される基礎となる **http-remoting connector** は、使用するセキュリティーレルムを決定します。**http-remoting connector** の詳細は、JBoss EAP **設定ガイド**の About the Remoting Subsystem を参照してください。

デフォルトコネクターのセキュリティーレルムは以下のように変更できます。

```
/subsystem=remoting/http-connector=http-remoting-connector:write-attribute(name=security-realm,value=MyDomainRealm)
```

### 6.4.5. SSL 暗号化を使用したリモート EJB アクセス

デフォルトでは、EJB2 および EJB3 Bean の Remote Method Invocation (RMI) のネットワークトラフィックは暗号化されません。暗号化が必要なインスタンスでは、クライアントとサーバー間の接続を暗号化する Secure Sockets Layer (SSL) を使用できます。また、SSL を使用すると、ファイアウォールの設定に応じて、ネットワークトラフィックが一部のファイアウォールを通過できるようになります。

**警告**

Red Hat では、影響するすべてのパッケージで TLSv1.1 または TLSv1.2 を利用するために SSLv2、SSLv3、および TLSv1.0 を明示的に無効化することを推奨しています。

## 6.5. EJB サブシステムとの ELYTRON の統合

JBoss EAP 7.1 より、デプロイメントをマッピングし、それらのセキュリティーが **elytron** サブシステムによって処理されるようになりました。デプロイメントがマップされたセキュリティードメインを参照する場合、そのセキュリティーは Elytron によって処理されます。そうでない場合には、そのセキュリティーはレガシーセキュリティーサブシステムによって処理されます。このマッピングは **ejb** サブシステムで定義されます。

**ejb** サブシステム内で、セキュリティードメイン名から、ディプロイメント内で参照されるものとして、参照される Elytron セキュリティードメインにマッピングが作成されます。マッピングされたセキュリティードメイン名がデプロイメント内の bean に対して設定されている場合は、セキュリティーが Elytron によって処理される必要があることを示します。既存の EJB セキュリティーインターセプターの代わりに新しい EJB セキュリティーインターセプターが設定されます。

新しい EJB セキュリティーインターセプターは、呼び出しに関連付けられた Elytron **SecurityDomain** を利用して現在の **SecurityIdentity** を取得し、以下のタスクを実行します。

- **run-as** プリンシパルを確立します。
- **run-as** プリンシパルの追加ロールを作成します 。
- **run-as** ロールを作成します。
- 承認の決定を行います。

JBoss EAP 7.1 では、**ejb** サブシステム **application-security-domains** に新しい管理リソースが導入されました。**application-security-domains** 要素には、Elytron セキュリティードメインにマップする必要のあるアプリケーションセキュリティードメインが含まれます。

**表6.1 application-security-domain の属性**

| 属性 | 説明 |
| --- | --- |
| **name** | この属性は、デプロイメントで指定されるセキュリティードメインの名前を参照します。 |
| **security-domain** | この属性は、使用される必要がある Elytron セキュリティードメインへの参照です。 |
| **enable-jacc** | この属性は、JACC を使用した承認を有効にします。 |
| **referencing-deployments** | これは、現在 ASD を参照しているすべてのデプロイメントを一覧表示するランタイム属性です。 |

以下の方法のいずれかで、**ejb** サブシステムで **application-security-domain** を設定できます。管理コンソールを使用 するか、管理 CLI を使用 できます。

### 6.5.1. 管理コンソールを使用したアプリケーションセキュリティードメインの設定

1. 管理コンソールにアクセスします。詳細は、JBoss EAP**設定ガイド**の 管理コンソール を参照してください。
2. **Configuration → Subsystems → EJB** の順に移動し、**View** をクリックします。
3. **Security Domain** タブを選択し、必要に応じてアプリケーションセキュリティードメインを設定します。

### 6.5.2. 管理 CLI を使用したアプリケーションセキュリティードメインの設定

以下の例では、**MyAppSecurity** はデプロイメントで参照されるセキュリティードメインであり、**ApplicationDomain** は **elytron** サブシステムに設定された Elytron セキュリティードメインです。

```
/subsystem=ejb3/application-security-domain=MyAppSecurity:add(security-
domain=ApplicationDomain)
```

以下の XML は、このコマンドの結果としてサーバー設定ファイルの **ejb** サブシステムに追加されます。

```
<application-security-domains>
    <application-security-domain name="MyAppSecurity" security-domain="ApplicationDomain"/>
</application-security-domains>
```

Elytron を使用してセキュリティーを処理する EJB の単純な作業例は、JBoss EAP に同梱される **ejb-security** クイックスタートを参照してください。

# 第7章 EJB インターセプター

## 7.1. カスタムインターセプター

JBoss EAP では、カスタムの EJB インターセプターを開発し、管理できます。

以下のタイプのインターセプターを作成できます。

- クライアントインターセプター
  クライアントインターセプターは、JBoss EAP がクライアントとして動作するときに実行されます。
- サーバーインターセプター
  サーバーインターセプターは、JBoss EAP がサーバーとして動作するときに実行されます。これらのインターセプターは、サーバーにグローバルに設定できます。
- コンテナーインターセプター
  コンテナーインターセプターは、JBoss EAP がサーバーとして動作するときに実行されます。これらのインターセプターは EJB コンテナーで設定されます。

カスタムインターセプタークラスをモジュールに追加し、**$JBOSS_HOME/modules** ディレクトリーに格納する必要があります。

### 7.1.1. インターセプターチェーン

カスタムインターセプターは、インターセプターチェーンの特定ポイントで実行されます。

EJB に設定されたコンテナーインターセプターは、セキュリティーインターセプターやトランザクション管理インターセプターなど、Wildfly が提供するインターセプターの前に実行されます。そのため、コンテナーインターセプターは、Wildfly インターセプターまたはグローバルインターセプターを起動する前にコンテキストデータを処理または設定できます。

サーバーインターセプターおよびクライアントインターセプターは、Wildfly 固有のインターセプターの後に実行されます。

### 7.1.2. カスタムクライアントインターセプター

カスタムクライアントインターセプターは、**org.jboss.ejb.client.EJBClientInterceptor** インターフェイスを実装します。

**Org.jboss.ejb.client.EJBClientInvocationContext** インターフェイスも含める必要があります。

以下のコードは、クライアントインターセプターの例を示しています。

**クライアントインターセプターコードの例**

```
package org.foo;
import org.jboss.ejb.client.EJBClientInterceptor;
import org.jboss.ejb.client.EJBClientInvocationContext;
public class FooInterceptor implements EJBClientInterceptor {
    @Override
    public void handleInvocation(EJBClientInvocationContext context) throws Exception {
        context.sendRequest();
    }
```

```
    @Override
    public Object handleInvocationResult(EJBClientInvocationContext context) throws Exception {
        return context.getResult();
    }
}
```

### 7.1.3. カスタムサーバーインターセプター

サーバーインターセプターは、**@javax.annotation.AroundInvoke** アノテーションまたは **javax.interceptor.AroundTimeout** アノテーションを使用して、Bean での呼び出し中に呼び出されるメソッドをマークします。

以下のコードは、サーバーインターセプターの例を示しています。

**サーバーインターセプターコードの例**

```
package org.testsuite.ejb.serverinterceptor;
import javax.annotation.PostConstruct;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;
public class TestServerInterceptor {
    @AroundInvoke
    public Object aroundInvoke(final InvocationContext invocationContext) throws Exception {

        return invocationContext.proceed();
    }
}
```

### 7.1.4. カスタムコンテナーインターセプター

コンテナーインターセプターは、**@javax.annotation.AroundInvoke** アノテーションまたは **javax.interceptor.AroundTimeout** アノテーションを使用して、Bean での呼び出し中に呼び出されるメソッドをマークします。

Jakarta Enterprise Beans 3.2 仕様で定義されている標準の JakartaEE インターセプターは、コンテナーがセキュリティーコンテキストの伝搬、トランザクション管理、およびその他コンテナーによる呼び出し処理の完了後に実行されることが予想されます。

以下のコードは、呼び出しのために **iAmAround** メソッドをマークするインターセプタークラスを示しています。

**コンテナーインターセプターコードの例**

```
public class ClassLevelContainerInterceptor {
    @AroundInvoke
    private Object iAmAround(final InvocationContext invocationContext) throws Exception {
        return this.getClass().getName() + " " + invocationContext.proceed();
    }
}
```

**コンテナーインターセプターと Jakarta EE インターセプター API の違い**
コンテナーインターセプターは Jakarta EE インターセプターと同様にモデル化されますが、API のセマンティクスにはいくつかの違いがあります。たとえば、これらのインターセプターは EJB コンポーネントの設定またはインスタンス化よりもだいぶ前に呼び出されるため、コンテナーインターセプターが

**javax.interceptor.InvocationContext.getTarget()** メソッドを呼び出すことは違反になります。

### 7.1.5. コンテナーインターセプターの設定

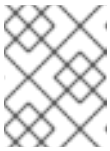
コンテナーインターセプターは標準の Jakarta EE インターセプターライブラリーを使用します。

そのため、ejb-jar デプロイメント記述子の 3.2 バージョンに対して **ejb-jar.xml** ファイルで許可されるのと同じ XSD 要素を使用します。

それらは標準の Jakarta EE インターセプターライブラリーをベースとしているため、コンテナーインターセプターはデプロイメント記述子を使用してのみ設定できます。設計アプリケーションには、JBoss EAP 固有のアノテーションやその他のライブラリーの依存関係は必要ありません。

コンテナーインターセプターを設定するには、以下を実行します。

1. EJB デプロイメントの **META-INF/** ディレクトリーに **jboss-ejb3.xml** ファイルを作成します。
2. 記述子ファイルにコンテナーインターセプター要素を設定します。
    a. **urn:container-interceptors:1.0** ネームスペースを使用して、コンテナーインターセプター要素の設定を指定します。
    b. **<container-interceptors>** 要素を使用してコンテナーインターセプターを指定します。
    c. **>interceptor-binding>** 要素を使用してコンテナーインターセプターを EJB にバインドします。インターセプターは、以下のいずれかの方法でバインドできます。
        - ワイルドカード (*) を使用して、インターセプターをデプロイメントのすべての EJB にバインドします。
        - 特定の EJB 名を使用して、個々の Bean レベルでインターセプターをバインドします。
        - EJB の特定のメソッドレベルでインターセプターをバインドします。



**注記**

これらの要素は、Jakarta EE インターセプターと同じ方法で EJB 3.2 XSD を使用して設定されます。

以下の記述子ファイル例は設定オプションを示しています。

**コンテナーインターセプター jboss-ejb3.xml ファイルの例**

```
<jboss xmlns="http://www.jboss.com/xml/ns/javaee"
      xmlns:jee="http://java.sun.com/xml/ns/javaee"
      xmlns:ci ="urn:container-interceptors:1.0">
   <jee:assembly-descriptor>
      <ci:container-interceptors>
         <!-- Default interceptor -->
         <jee:interceptor-binding>
            <ejb-name>*</ejb-name>
            <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.ContainerInterceptorOne</interceptor-
class>
         </jee:interceptor-binding>
```

```
        <!-- Class level container-interceptor -->
        <jee:interceptor-binding>
          <ejb-name>AnotherFlowTrackingBean</ejb-name>
          <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.ClassLevelContainerInterceptor</intercept
or-class>
        </jee:interceptor-binding>
        <!-- Method specific container-interceptor -->
        <jee:interceptor-binding>
          <ejb-name>AnotherFlowTrackingBean</ejb-name>
          <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.MethodSpecificContainerInterceptor</inte
rceptor-class>
            <method>
              <method-name>echoWithMethodSpecificContainerInterceptor</method-name>
            </method>
        </jee:interceptor-binding>
        <!-- container interceptors in a specific order -->
        <jee:interceptor-binding>
          <ejb-name>AnotherFlowTrackingBean</ejb-name>
          <interceptor-order>
            <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.ClassLevelContainerInterceptor</intercept
or-class>
            <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.MethodSpecificContainerInterceptor</inte
rceptor-class>
            <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.ContainerInterceptorOne</interceptor-
class>
          </interceptor-order>
          <method>
            <method-name>echoInSpecificOrderOfContainerInterceptors</method-name>
          </method>
        </jee:interceptor-binding>
      </ci:container-interceptors>
    </jee:assembly-descriptor>
</jboss>
```

**allow-ejb-name-regex** 属性を使用すると、インターセプターバインディングで正規表現を使用できるようになり、指定の正規表現に一致するすべての Bean にインターセプターをマッピングできます。以下の管理 CLI コマンドを使用して、**ejb3** サブシステムの **allow-ejb-name-regex** 属性を **true** に有効化します。

```
/subsystem=ejb3:write-attribute(name=allow-ejb-name-regex,value=true)
```

**urn:container-interceptors:1.0** 名前空間のスキーマは http://www.jboss.org/schema/jbossas/jboss-ejb-container-interceptors_1_0.xsd で利用できます。

### 7.1.6. サーバーおよびクライアントインターセプターの設定

サーバーインターセプターおよびクライアントインターセプターは、使用される設定ファイルの JBoss EAP 設定にグローバルに追加されます。

サーバーインターセプターは、**ejb3** サブシステム設定の **<server-interceptors>** 要素に追加されます。クライアントインターセプターは、**ejb3** サブシステム設定の **<client-interceptors>** 要素に追加されます。

以下の例は、サーバーインターセプターの追加を示しています。

```
/subsystem=ejb3:list-add(name=server-interceptors,value={module=org.abccorp:tracing-
interceptors:1.0,class=org.abccorp.TracingInterceptor})
```

以下の例は、クライアントインターセプターの追加を示しています。

```
/subsystem=ejb3:list-add(name=client-interceptors,value=
{module=org.abccorp:clientInterceptor:1.0,class=org.abccorp.clientInterceptor})
```

サーバーインターセプターまたはクライアントインターセプターが追加されるか、インターセプターの設定が変更されるたびに、サーバーをリロードする必要があります。

### 7.1.7. SCC (Security Context Identity) の変更

複数のクライアント接続を開く代わりに、認証されたユーザーにアイデンティティーを切り替え、別のユーザーとして既存の接続で要求を実行するパーミッションを付与することができます。

デフォルトでは、アプリケーションサーバーにデプロイされた EJB へのリモート呼び出しを行うと、サーバーへの接続は認証され、接続を使用する後続のリクエストは元の認証済みアイデンティティーを使用して実行されます。これは、クライアントとサーバー間の呼び出しの両方に適用されます。同じクライアントから異なるアイデンティティーを使用する必要がある場合は、通常は、異なるアイデンティティーとして認証されるように、サーバーへの接続を複数開く必要があります。代わりに、認証されたユーザーがアイデンティティーを変更できるようにすることもできます。

認証されたユーザーのアイデンティティーを変更するには、以下を実行します。

1. インターセプターコードでアイデンティティーの変更を実装します。

   - クライアントインターセプター
     インターセプターは、**EJBClientInvocationContext.getContextData()** への呼び出しを使用して取得できるコンテキストデータマップを介して要求されたアイデンティティーを渡す必要があります。以下のサンプルコードは、アイデンティティーを切り替えるクライアントインターセプターを示しています。

     **クライアントインターセプターコードの例**

```
public class ClientSecurityInterceptor implements EJBClientInterceptor {

   public void handleInvocation(EJBClientInvocationContext context) throws Exception {
      Principal currentPrincipal = SecurityActions.securityContextGetPrincipal();

      if (currentPrincipal != null) {
         Map<String, Object> contextData = context.getContextData();
         contextData.put(ServerSecurityInterceptor.DELEGATED_USER_KEY,
currentPrincipal.getName());
      }
      context.sendRequest();
   }

   public Object handleInvocationResult(EJBClientInvocationContext context) throws
```

```
Exception {
        return context.getResult();
    }
}
```

- コンテナーおよびサーバーインターセプター
  これらのインターセプターはアイデンティティーが含まれる **InvocationContext** を受け取り、新しいアイデンティティーに切り替える要求を行います。以下のコードは、コンテナーインターセプターのブリッジした例を示しています。

  **コンテナーインターセプターコードの例**

```
public class ServerSecurityInterceptor {

    private static final Logger logger = Logger.getLogger(ServerSecurityInterceptor.class);

    static final String DELEGATED_USER_KEY =
ServerSecurityInterceptor.class.getName() + ".DelegationUser";

    @AroundInvoke
    public Object aroundInvoke(final InvocationContext invocationContext) throws
Exception {
        Principal desiredUser = null;
        UserPrincipal connectionUser = null;

        Map<String, Object> contextData = invocationContext.getContextData();
        if (contextData.containsKey(DELEGATED_USER_KEY)) {
            desiredUser = new SimplePrincipal((String)
contextData.get(DELEGATED_USER_KEY));

            Collection<Principal> connectionPrincipals =
SecurityActions.getConnectionPrincipals();

            if (connectionPrincipals != null) {
                for (Principal current : connectionPrincipals) {
                    if (current instanceof UserPrincipal) {
                        connectionUser = (UserPrincipal) current;
                        break;
                    }
                }

            } else {
                throw new IllegalStateException("Delegation user requested but no user on
connection found.");
            }
        }

        ContextStateCache stateCache = null;
        try {
            if (desiredUser != null && connectionUser != null
                && (desiredUser.getName().equals(connectionUser.getName()) == false)) {
                // The final part of this check is to verify that the change does actually indicate a
change in user.
                try {
                    // We have been requested to use an authentication token
                    // so now we attempt the switch.
```

```
                stateCache = SecurityActions.pushIdentity(desiredUser, new
OuterUserCredential(connectionUser));
            } catch (Exception e) {
                logger.error("Failed to switch security context for user", e);
                // Don't propagate the exception stacktrace back to the client for security
reasons
                throw new EJBAccessException("Unable to attempt switching of user.");
            }
        }

        return invocationContext.proceed();
    } finally {
        // switch back to original context
        if (stateCache != null) {
            SecurityActions.popIdentity(stateCache);;
        }
    }
}
```

2. アプリケーションは、プログラムを用いて、またはサービス出力ダーメカニズムを使用して、クライアントインターセプターを **EJBClientContext** インターセプターチェーンに挿入できます。クライアントインターセプターを設定する手順は、Using a Client Interceptor in an Application を参照してください。

3. Jakarta Authentication ログインモジュールを作成します。
Jakarta Authentication LoginModule コンポーネントは、ユーザーが要求されたアイデンティティーとして要求を実行できることを確認します。以下の abridged コード例は、ログインおよび検証を実行するメソッドを示しています。

**LoginModule コードの例**

```
@SuppressWarnings("unchecked")
@Override
public boolean login() throws LoginException {
    if (super.login() == true) {
        log.debug("super.login()==true");
        return true;
    }

    // Time to see if this is a delegation request.
    NameCallback ncb = new NameCallback("Username:");
    ObjectCallback ocb = new ObjectCallback("Password:");

    try {
        callbackHandler.handle(new Callback[] { ncb, ocb });
    } catch (Exception e) {
        if (e instanceof RuntimeException) {
            throw (RuntimeException) e;
        }
        // If the CallbackHandler can not handle the required callbacks then no chance.
        return false;
    }

    String name = ncb.getName();
    Object credential = ocb.getCredential();
```

```
        if (credential instanceof OuterUserCredential) {
            // This credential type will only be seen for a delegation request, if not seen then the
request is not for us.

            if (delegationAcceptable(name, (OuterUserCredential) credential)) {
                identity = new SimplePrincipal(name);
                if (getUseFirstPass()) {
                    String userName = identity.getName();
                    if (log.isDebugEnabled())
                        log.debug("Storing username '" + userName + "' and empty password");
                    // Add the username and an empty password to the shared state map
                    sharedState.put("javax.security.auth.login.name", identity);
                    sharedState.put("javax.security.auth.login.password", "");
                }
                loginOk = true;
                return true;
            }
        }
        return false; // Attempted login but not successful.
    }

    // Make a trust user to decide if the user switch is acceptable.
    protected boolean delegationAcceptable(String requestedUser, OuterUserCredential
connectionUser) {
    if (delegationMappings == null) {
        return false;
    }

    String[] allowedMappings = loadPropertyValue(connectionUser.getName(),
connectionUser.getRealm());
    if (allowedMappings.length == 1 && "*".equals(allowedMappings[0])) {
        // A wild card mapping was found.
        return true;
    }
    for (String current : allowedMappings) {
        if (requestedUser.equals(current)) {
            return true;
        }
    }
    return false;
}
```

### 7.1.8. アプリケーションでのクライアントインターセプターの使用

アプリケーションは、プログラムを用いて、またはサービス出力ダーメカニズムを使用して、あるいは、ClientInterceptors アノテーションを使用して、クライアントインターセプターを **EJBClientContext** インターセプターチェーンに挿入できます。

**注記**

**EJBClientInterceptor** は、**org.jboss.ejb.client.EJBClientInvocationContext#addReturnedContextDataKey(String key)** を呼び出してサーバー側の呼び出しコンテキストから特定のデータをリクエストすることができます。要求したデータがコンテキストデータマップの提供されるキー以下に存在する場合、クライアントに送信されます。

#### 7.1.8.1. クライアントインターセプターのプログラムでの挿入

インターセプターを登録した **EJBClientContext** を作成したら、インターセプターを挿入します。

以下のコードは、インターセプターの登録で **EJBClientContext** を作成する方法を示しています。

```
EJBClientContext ctxWithInterceptors =
EJBClientContext.getCurrent().withAddedInterceptors(clientInterceptor);
```

**EJBClientContext** の作成後、インターセプターを挿入するオプションを使用できます。

- 以下のコードは、**Callable** 操作を使用して **EJBClientContext** を適用して実行できます。**Callable** 操作内で実行される EJB 呼び出しはクライアント側のインターセプターを適用します。

```
ctxWithInterceptors.runCallable(() -> {
    // perform the calls which should use the interceptor
})
```

- または、新規作成された **EJBClientContext** を新しいデフォルトとしてマークすることもできます。

```
EJBClientContext.getContextManager().setThreadDefault(ctxWithInterceptors);
```

#### 7.1.8.2. サービ出力ダーメカニズムを使用したクライアントインターセプターの挿入

**META-INF/services/org.jboss.ejb.client.EJBClientInterceptor** ファイルを作成し、クライアントアプリケーションのクラスパスに配置またはパッケージ化します。

ファイルのルールは、Java ServiceLoader Mechanism によって指示されます。

- このファイルには、EJB クライアントインターセプター実装の完全修飾クラス名ごとに個別の行が含まれることが予想されます。
- EJB クライアントインターセプタークラスはクラスパスで利用できる必要があります。

サービ出力ダーメカニズムを使用して追加された EJB クライアントインターセプターは、クラスパスにある順序で追加され、クライアントインターセプターチェーンの最後に追加されます。

#### 7.1.8.3. ClientInterceptor アノテーションを使用したクライアントインターセプターの挿入

**@org.jboss.ejb.client.annotation.ClientInterceptors** アノテーションを使用すると、EJB インターセプターをリモート呼び出しのクライアント側に配置できます。

```
import org.jboss.ejb.client.annotation.ClientInterceptors;
@ClientInterceptors({HelloClientInterceptor.class})

public interface HelloBeanRemote {
   public String hello();
}
```

# 第8章 CLUSTERED ENTERPRISE JAVABEANS (EJB)

## 8.1. クラスター化された EJB について

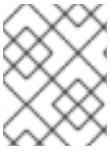
高可用性シナリオの EJB コンポーネントはクラスター化できます。HTTP コンポーネントとは異なるプロトコルを使用するため、複数の方法でクラスター化されます。EJB 2 および 3 のステートフル Bean およびステートレス Bean はクラスター化できます。

シングルトンの詳細は JBoss EAP開発ガイド の**HA シングルトンサービス** を参照してください。

## 8.2. EJB クライアントコード単純化

EJB サーバー側クラスターコンポーネントを呼び出す際に、EJB クライアントコードを簡素化できます。以下の手順では、EJB クライアントコードを単純化する複数の方法を概説します。

- 初期コンテキストルックアップ
- リモート EJB 設定ファイル
- EJB の自動トランザクション Stickiness
- クラスター環境の EJB トランザクション



**注記**

**jboss-ejb-client.properties** ファイルの使用は非推奨となり、**wildfly-config.xml** ファイルが優先されるようになりました。

## 8.3. クラスター化された EJB のデプロイ

クラスターリングサポートは JBoss EAP 7.3 の HA プロファイルで利用できます。HA 機能が有効なスタンドアロンサーバーを起動するには、**standalone-ha.xml** または **standalone-full-ha.xml** ファイルで起動します。

```
$ EAP_HOME/bin/standalone.sh --server-config=standalone-ha.xml
```

これにより、HA 機能を持つサーバーの単一インスタンスが起動します。

クラスターリングの利点を確認できるようにするには、サーバーの複数のインスタンスが必要になります。したがって、HA 機能を持つ別のサーバーを起動します。サーバーの別のインスタンスは、同じマシンまたは別のマシンのいずれかに置くことができます。同じマシン上にある場合は、以下の点に注意する必要があります。

- 次のインスタンスのポートオフセットを渡します。
- 各サーバーインスタンスに固有の **jboss.node.name** システムプロパティーがあることを確認します。

これは、以下のシステムプロパティーを起動コマンドに渡すことで実行できます。

```
$ EAP_HOME/bin/standalone.sh --server-config=standalone-ha.xml -Djboss.socket.binding.port-offset=PORT_OFFSET -Djboss.node.name=UNIQUE_NODE_NAME
```

EJB デプロイメントをこのインスタンスにデプロイするのにも慣れていることに気付かれる方法にしたがいます。

**警告**

クラスター化されたサーバーのスタンドアロンインスタンスの単一のノードにアプリケーションをデプロイしても、そのアプリケーションが他のクラスターインスタンスに自動的にデプロイされるわけではありません。これは、他のスタンドアロンクラスターインスタンスにも明示的にデプロイする必要があります。または、ドメインモードでサーバーを起動して、デプロイメントをサーバーグループのすべてのサーバーにデプロイできます。

クラスター化された EJB を持つアプリケーションを両方のインスタンスにデプロイしたことで、EJB はクラスターリング機能を活用できるようになりました。

**注記**

JBoss EAP 7 より、JBoss EAP が HA プロファイルを使用して起動されると、ステートフルセッション bean の状態がレプリケートされます。クラスターリング動作を有効にするために **@Clustered** アノテーションを使用する必要がなくなりました。

**@Stateful** アノテーションで **passivationCapable** を **false** に設定することで、ステートフルセッション Bean のレプリケーションを無効にできます。

```
@Stateful(passivationCapable=false)
```

これは、**cache-ref** ではなく **passivation-disabled-cache-ref** で定義された **ejb** キャッシュを使用するようサーバーに指示します。

ステートフルセッション Bean のレプリケーションをグローバルに無効にするには、以下の管理 CLI コマンドを使用します。

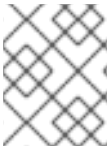
```
/subsystem=ejb3:write-attribute(name=default-sfsb-cache,value=simple)
```

## 8.4. クラスター化された EJB のフェイルオーバー

クラスター化された EJB にはフェイルオーバー機能があります。**@Stateful** EJB の状態がクラスターノード間で複製されるため、クラスター内のいずれかのノードがダウンした場合に、その他のノードが呼び出しを引き継ぐことができます。

クラスター内のサーバーがクラッシュした場合など、クラスター環境の状況によっては、EJB クライアントは応答の代わりに例外を受信する場合があります。EJB クライアントライブラリーは、発生する障害のタイプに応じて、安全に呼び出しを再試行します。ただし、要求が失敗し、再試行の安全性を確保できない場合は、ご自分の環境に合わせて例外を処理できます。ただし、カスタムのインターセプターを使用して、再試行動作を追加することもできます。

## 8.5. リモートスタンドアロンクライアント



**注記**

**jboss-ejb-client.properties** ファイルの使用は非推奨となり、**wildfly-config.xml** ファイルが優先されるようになりました。

スタンドアロンリモートクライアントは、Java Naming and Directory Interface のアプローチまたはネイティブ JBoss EJB クライアント API を使用してサーバーと通信できます。重要なことは、クラスター化された EJB デプロイメントを呼び出すときに、クラスター内のすべてのサーバーを一覧表示する必要がないことです。これは、クラスター内でのクラスターノードの追加の動的な性質があるため、実行できません。

リモートクライアントは、クラスターリング機能を持ついずれかのサーバーのみを一覧表示する必要があります。このサーバーは、クライアントとクラスター化されたノード間のクラストートポロジー通信の開始点として機能します。

**jboss-ejb-client.properties** 設定ファイルで **ejb** クラスターを設定する必要があります。

```
remote.clusters=ejb
remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED=false
```

## 8.6. クラストートポロジー通信

**注記**

**jboss-ejb-client.properties** ファイルの使用は非推奨となり、**wildfly-config.xml** ファイルが優先されるようになりました。

JBoss EJB クライアント実装は、クライアントがサーバーに接続すると、サーバーにクラスターリング機能がある場合にクラストートポロジー情報のサーバーの内部と通信します。たとえば、接続する最初のサーバーとして サーバー X が一覧表示される場合は、クライアントがサーバー X に接続すると、そのサーバーは非同期クラストートポロジーメッセージをクライアントに送信します。このトポロジーメッセージは、クラスター名と、クラスターに属するノードの情報で設定されます。ノード情報には、必要に応じて接続するノードアドレスおよびポート番号が含まれます。この例では、サーバー X はクラスターに属する他のサーバー Y で設定されるクラストートポロジーを返します。

ステートフルクラスター EJB の場合、呼び出しフローは 2 つのステップで生じます。

1. Bean の Java Naming および Directory Interface ルックアップを実行するときに発生するステートフル Bean のセッションの作成。

2. 返されたプロキシーの呼び出し。

ステートフル Bean のルックアップは、内部的に、クライアントからサーバーへの同期セッション作成リクエストをトリガーします。この場合、**jboss-ejb-client.properties** ファイルで設定されたため、セッション作成リクエストはサーバー X に送信されます。サーバー X はクラスター化されているため、セッション ID を返し、そのセッションの**アフィニティー**を送り返します。クラスター化サーバーの場合、**アフィニティー**はステートフル Bean がサーバー側にあるクラスターの名前と同じになります。クラスター化されていない Bean の場合、アフィニティーはセッションが作成されたノード名です。この **アフィニティー**は、EJB クライアントがクラスター化 Bean のクラスター内のノード、または非クラスター Bean の特定のノードにプロキシー上の呼び出しをルーティングするのに役立ちます。このセッション作成リクエストが発生しますが、サーバー X もクラストートポロジーが含まれる非同期

メッセージを送信します。JBoss EJB クライアント実装は、このトポロジー情報を記録し、後でクラスター内のノードへの接続作成に使用し、必要に応じてそれらのノードへの呼び出しをルーティングします。

フェイルオーバーの仕組みを理解するには、開始点と同じサーバー X の例と、ステートフル Bean を検索し、呼び出しているクライアントアプリケーションを検討してください。これらの呼び出し中に、クライアント側はサーバーからクラスタートポロジー情報を収集します。何らかの理由でサーバー X が停止し、クライアントアプリケーションがプロキシーで呼び出されることを仮定します。この段階で JBoss EJB クライアント実装は**アフィニティー**を認識する必要があり、この場合はクラスターのアフィニティーになります。クライアントが持つクラスタートポロジー情報から、クラスターにはサーバー X とサーバー Y のノードがあることを認識します。呼び出しが完了すると、クライアントはサーバー X が停止していることを認識できるため、セレクターを使用してクラスターノードから適切なノードをフェッチします。セレクターがクラスターノードからノードを返す場合、JBoss EJB クライアント実装は、接続がすでに作成されていない場合にそのノードへの接続を作成し、そこから EJB レシーバーを作成します。この例では、クラスターの他の唯一のノードはサーバー Y なので、セレクターはサーバー Y をノードとして返し、JBoss EJB クライアント実装はそれを使用して EJB レシーバーを作成し、このレシーバーを使用してプロキシーの呼び出しを渡します。実際には、呼び出しがクラスター内の別のノードにフェイルオーバーするようになりました。

## 8.7. EJB の自動トランザクション STICKINESS

EJB プロキシーと同じコンテキストからルックアップされるトランザクションオブジェクトは、同じホストをターゲットとします。アクティブトランザクションを使用すると、コンテキストがマルチホストである場合や、クラスター化されている場合は、呼び出しコンテキストが同じノードに固定されます。

この動作は、トランザクションをアウトフローしたか、リモートユーザートランザクションを使用しているかによって異なります。

アウトフロートランザクションの場合、アプリケーションが特定のノードでルックアップされると、同じトランザクション下にあるアプリケーションへの呼び出しすべてが、このノードを対象にしようとします。アウトフローされたトランザクションをすでに受信したノードは、まだ受信されていないノードよりも優先されます。

リモートユーザートランザクションでは、最初に成功した呼び出しによって指定のノードにトランザクションがロックされ、このトランザクション下の後続の呼び出しは同じノードに移動する必要があり、そうでない場合は例外が発生します。

## 8.8. 別のインスタンス上のリモートクライアント

ここでは、JBoss EAP インスタンスにデプロイされたクライアントアプリケーションが、別の JBoss EAP インスタンスにデプロイされた、クラスター化されたステートフル Bean を呼び出す方法を説明します。

以下の例では、3 つのサーバーが関与しています。サーバー X と Y の両方がクラスターに属し、クラスター化された EJB がクラスター内にデプロイされます。他のサーバーインスタンスサーバー C もあり、クラスターリング機能も持たない可能性があります。サーバー C は、サーバー X および Y にデプロイされたクラスター化された Bean を起動し、フェイルオーバーを達成するデプロイメントがあるクライアントとして機能します。

この設定は **jboss-ejb-client.xml** ファイルで行われます。このファイルは、他のサーバーへのリモートアウトバウンド接続を参照します。サーバー C はクライアントであるため、**jboss-ejb-client.xml** ファイルの設定はサーバー C のデプロイメントにあります。クライアント設定はすべてのクラスター化されたノードを参照する必要はなく、いずれか 1 つを参照します。これは、通信の開始点として機能します。

この場合、リモートアウトバウンド接続はサーバー C からサーバー X に作成され、次にサーバー X が通信の開始点として使用されます。リモートスタンドアロンクライアントの場合と同様に、サーバー C のアプリケーションがステートフル Bean をルックアップすると、セッション ID と、そのクラスターのアフィニティーを返すサーバー X にセッション作成要求が送信されます。また、サーバー X は、クラスタートポロジーが含まれるサーバー C に非同期メッセージを送信します。サーバー Y はサーバー X とともにクラスターに属するため、このトポロジー情報にはサーバー Y のノード情報が含まれます。プロキシーの後続の呼び出しはクラスター内のノードに適切にルーティングされます。前述のようにサーバー X がダウンした場合は、クラスターからの別のノードが選択され、呼び出しはそのノードに転送されます。

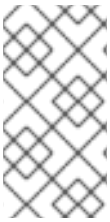
フェイルオーバーの観点では、リモートスタンドアロンクライアントと別の JBoss EAP インスタンス上のリモートクライアントの両方が同じように動作します。

## 8.9. スタンドアロンおよびサーバー内クライアントの設定

**注記**

**jboss-ejb-client.properties** ファイルの使用は非推奨となり、**wildfly-config.xml** ファイルが優先されるようになりました。

EJB クライアントをクラスター化された EJB アプリケーションに接続するには、スタンドアロン EJB クライアントまたはサーバー内 EJB クライアントの既存の設定を拡張して、クラスター接続設定を組み込む必要があります。スタンドアロン EJB クライアントの **jboss-ejb-client.properties**、またはサーバー側のアプリケーションの **jboss-ejb-client.xml** ファイルを拡張してクラスター設定を組み込む必要があります。



**注記**

EJB クライアントは、リモートサーバーで EJB を使用するプログラムです。リモートサーバーを呼び出す **EJB** クライアント自体がサーバー内で実行されると、クライアントはサーバー内で実行されます。つまり、別の JBoss EAP インスタンスを呼び出す JBoss EAP インスタンスは、サーバー内クライアントとみなされます。

以下の例は、スタンドアロン EJB クライアントに必要な追加のクラスター設定を示しています。

```
remote.clusters=ejb
remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED=false
remote.cluster.ejb.username=test
remote.cluster.ejb.password=password
```

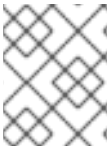
アプリケーションが remote-outbound-connection を使用する場合は、以下の例のように **jboss-ejb-client.xml** ファイルを設定し、クラスター設定を追加する必要があります。

```
<jboss-ejb-client xmlns:xsi="urn:jboss:ejb-client:1.2" xsi:noNamespaceSchemaLocation="jboss-ejb-client_1_2.xsd">
  <client-context>
    <ejb-receivers>
      <!-- this is the connection to access the app-one -->
      <remoting-ejb-receiver outbound-connection-ref="remote-ejb-connection-1" />
      <!-- this is the connection to access the app-two -->
      <remoting-ejb-receiver outbound-connection-ref="remote-ejb-connection-2" />
    </ejb-receivers>
```

```
    <!-- If an outbound connection connects to a cluster,
            a list of members is provided after successful connection.
         To connect to this node this cluster element must be defined. -->

    <clusters>
      <!-- cluster of remote-ejb-connection-1 -->
      <cluster name="ejb" security-realm="ejb-security-realm-1" username="quickuser1">
        <connection-creation-options>
        <property name="org.xnio.Options.SSL_ENABLED" value="false" />
        <property name="org.xnio.Options.SASL_POLICY_NOANONYMOUS" value="false" />
        </connection-creation-options>
      </cluster>
    </clusters>
  </client-context>
</jboss-ejb-client>
```

remote-outbound-connection の詳細は、JBoss EAP 設定ガイド の **About Remoting Subsystem** を参照してください。



**注記**

セキュアな接続では、認証例外を回避するために認証情報をクラスター設定に追加する必要があります。

## 8.10. EJB 呼び出しのカスタムロードバランシングポリシーの実装

**注記**

**jboss-ejb-client.properties** ファイルの使用は非推奨となり、**wildfly-config.xml** ファイルが優先されるようになりました。

アプリケーションの EJB 呼び出しをサーバー間で分散するために、代替またはカスタマイズしたロードバランシングポリシーを実装することができます。

EJB 呼び出しに **AllClusterNodeSelector** を実装することができます。**AllClusterNodeSelector** のノード選択の動作はデフォルトのセレクターと似ていますが、大規模なクラスター (ノード数 > 20) の場合でも **AllClusterNodeSelector** は利用可能なすべてのクラスターノードを使用する点が異なります。接続されていないクラスターノードが返されると、そのクラスターノードは自動的に開きます。以下の例は、**AllClusterNodeSelector** 実装を示しています。

```
package org.jboss.as.quickstarts.ejb.clients.selector;

import java.util.Arrays;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

import org.jboss.ejb.client.ClusterNodeSelector;
public class AllClusterNodeSelector implements ClusterNodeSelector {
  private static final Logger LOGGER = Logger.getLogger(AllClusterNodeSelector.class.getName());

  @Override
  public String selectNode(final String clusterName, final String[] connectedNodes, final String[]
```

```
availableNodes) {
    if(LOGGER.isLoggable(Level.FINER)) {
      LOGGER.finer("INSTANCE "+this+ " : cluster:"+clusterName+"
connected:"+Arrays.deepToString(connectedNodes)+"
available:"+Arrays.deepToString(availableNodes));
    }

    if (availableNodes.length == 1) {
        return availableNodes[0];
    }
    final Random random = new Random();
    final int randomSelection = random.nextInt(availableNodes.length);
    return availableNodes[randomSelection];
  }

}
```

EJB 呼び出しの **SimpleLoadFactorNodeSelector** を実装することもできます。**SimpleLoadFactorNodeSelector** での負荷分散は、負荷係数に基づいて実行されます。負荷係数 (2/3/4) は、各ノードの負荷に関係なく、ノードの名前 (A/B/C) を基に計算されます。以下の例は、**SimpleLoadFactorNodeSelector** 実装を示しています。

```
package org.jboss.as.quickstarts.ejb.clients.selector;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

import org.jboss.ejb.client.DeploymentNodeSelector;
public class SimpleLoadFactorNodeSelector implements DeploymentNodeSelector {
  private static final Logger LOGGER =
Logger.getLogger(SimpleLoadFactorNodeSelector.class.getName());
  private final Map<String, List<String>[]> nodes = new HashMap<String, List<String>[]>();
  private final Map<String, Integer> cursor = new HashMap<String, Integer>();

  private ArrayList<String> calculateNodes(Collection<String> eligibleNodes) {
    ArrayList<String> nodeList = new ArrayList<String>();

    for (String string : eligibleNodes) {
      if(string.contains("A") || string.contains("2")) {
        nodeList.add(string);
        nodeList.add(string);
      } else if(string.contains("B") || string.contains("3")) {
        nodeList.add(string);
        nodeList.add(string);
        nodeList.add(string);
      } else if(string.contains("C") || string.contains("4")) {
        nodeList.add(string);
        nodeList.add(string);
        nodeList.add(string);
        nodeList.add(string);
```

```
      }
    }
    return nodeList;
  }

  @SuppressWarnings("unchecked")
  private void checkNodeNames(String[] eligibleNodes, String key) {
    if(!nodes.containsKey(key) || nodes.get(key)[0].size() != eligibleNodes.length || !nodes.get(key)
[0].containsAll(Arrays.asList(eligibleNodes))) {
      // must be synchronized as the client might call it concurrent
      synchronized (nodes) {
        if(!nodes.containsKey(key) || nodes.get(key)[0].size() != eligibleNodes.length || !nodes.get(key)
[0].containsAll(Arrays.asList(eligibleNodes))) {
          ArrayList<String> nodeList = new ArrayList<String>();
          nodeList.addAll(Arrays.asList(eligibleNodes));

          nodes.put(key, new List[] { nodeList, calculateNodes(nodeList) });
        }
      }
    }
  }
   private synchronized String nextNode(String key) {
    Integer c = cursor.get(key);
    List<String> nodeList = nodes.get(key)[1];

    if(c == null || c >= nodeList.size()) {
      c = Integer.valueOf(0);
    }

    String node = nodeList.get(c);
    cursor.put(key, Integer.valueOf(c + 1));

    return node;
  }

  @Override
  public String selectNode(String[] eligibleNodes, String appName, String moduleName, String
distinctName) {
    if (LOGGER.isLoggable(Level.FINER)) {
      LOGGER.finer("INSTANCE " + this + " : nodes:" + Arrays.deepToString(eligibleNodes) + "
appName:" + appName + " moduleName:" + moduleName
          + " distinctName:" + distinctName);
    }

    // if there is only one there is no sense to choice
    if (eligibleNodes.length == 1) {
      return eligibleNodes[0];
    }
    final String key = appName + "|" + moduleName + "|" + distinctName;

    checkNodeNames(eligibleNodes, key);
    return nextNode(key);
  }
}
```

### jboss-ejb-client.properties ファイルの設定

**remote.cluster.ejb.clusternode.selector** プロパティーを実装クラスの名前 (**AllClusterNodeSelector** または **SimpleLoadfactorNodeSelector**) とともに追加する必要があります。セレクターは、呼び出し時に利用可能な設定済みのサーバーをすべて表示します。以下の例では、**AllClusterNodeSelector** をクラスターノードセレクターとして使用します。

```
remote.clusters=ejb
remote.cluster.ejb.clusternode.selector=org.jboss.as.quickstarts.ejb.clients.selector.AllClusterNodeSelec
or
remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED=false
remote.cluster.ejb.username=test
remote.cluster.ejb.password=password

remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=one,two
remote.connection.one.host=localhost
remote.connection.one.port = 8080
remote.connection.one.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
remote.connection.one.username=user
remote.connection.one.password=user123
remote.connection.two.host=localhost
remote.connection.two.port = 8180
remote.connection.two.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

### EJB クライアント API の使用

プロパティー **remote.cluster.ejb.clusternode.selector** を **PropertiesBasedEJBClientConfiguration** コンストラクターの一覧に追加する必要があります。以下の例では、**AllClusterNodeSelector** をクラスターノードセレクターとして使用します。

```
Properties p = new Properties();
p.put("remote.clusters", "ejb");
p.put("remote.cluster.ejb.clusternode.selector",
"org.jboss.as.quickstarts.ejb.clients.selector.AllClusterNodeSelector");
p.put("remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS",
"false");
p.put("remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED", "false");
p.put("remote.cluster.ejb.username", "test");
p.put("remote.cluster.ejb.password", "password");

p.put("remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED", "false");
p.put("remote.connections", "one,two");
p.put("remote.connection.one.port", "8080");
p.put("remote.connection.one.host", "localhost");
p.put("remote.connection.two.port", "8180");
p.put("remote.connection.two.host", "localhost");

EJBClientConfiguration cc = new PropertiesBasedEJBClientConfiguration(p);
ContextSelector<EJBClientContext> selector = new ConfigBasedEJBClientContextSelector(cc);
EJBClientContext.setSelector(selector);

p = new Properties();
p.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
InitialContext context = new InitialContext(p);
```

### jboss-ejb-client.xml ファイルの設定

サーバー間の通信に負荷分散ポリシーを使用するには、クラスをアプリケーションとともにパッケージ化し、**META-INF** フォルダーにある **jboss-ejb-client.xml** で設定します。以下の例では、**AllClusterNodeSelector** をクラスターノードセレクターとして使用します。

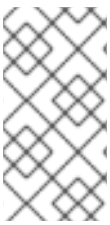
```
<jboss-ejb-client xmlns:xsi="urn:jboss:ejb-client:1.2" xsi:noNamespaceSchemaLocation="jboss-ejb-client_1_2.xsd">
  <client-context deployment-node-selector="org.jboss.ejb.client.DeploymentNodeSelector">
    <ejb-receivers>
      <!-- This is the connection to access the application. -->
      <remoting-ejb-receiver outbound-connection-ref="remote-ejb-connection-1" />
    </ejb-receivers>
    <!-- Specify the cluster configurations applicable for this client context -->
    <clusters>
      <!-- Configure the cluster of remote-ejb-connection-1. -->
      <cluster name="ejb" security-realm="ejb-security-realm-1" username="test" cluster-node-selector="org.jboss.as.quickstarts.ejb.clients.selector.AllClusterNodeSelector">
        <connection-creation-options>
          <property name="org.xnio.Options.SSL_ENABLED" value="false" />
          <property name="org.xnio.Options.SASL_POLICY_NOANONYMOUS" value="false" />
        </connection-creation-options>
      </cluster>
    </clusters>
  </client-context>
</jboss-ejb-client>
```

上記のセキュリティー設定を使用するには、client-server 設定に **ejb-security-realm-1** を追加する必要があります。以下の例は、セキュリティーレルムを追加する CLI コマンド (**ejb-security-realm-1**) を示しています。値は、ユーザー "test" の base64 でエンコードされたパスワードです。

```
/core-service=management/security-realm=ejb-security-realm-1:add()
/core-service=management/security-realm=ejb-security-realm-1/server-identity=secret:add(value=cXVpY2sxMjMr)
```

サーバー間の通信に負荷分散ポリシーを使用する場合は、クラスをアプリケーションとともにパッケージ化するか、モジュールとしてパッケージ化できます。このクラスは、トップレベルの EAR アーカイブの **META-INF** ディレクトリーにある **jboss-ejb-client** 設定ファイルで設定されます。以下の例では **RoundRobinNodeSelector** をデプロイメントノードセレクターとして使用します。

```
<jboss-ejb-client xmlns="urn:jboss:ejb-client:1.2">
    <client-context deployment-node-selector="org.jboss.example.RoundRobinNodeSelector">
        <ejb-receivers>
            <remoting-ejb-receiver outbound-connection-ref="..."/>
        </ejb-receivers>
        ...
    </client-context>
</jboss-ejb-client>
```



**注記**

スタンドアロンサーバーを実行している場合は、サーバー名の設定にオプション **-Djboss.node.name=** またはサーバー設定ファイル **standalone.xml** を使用します。サーバー名が一意であることを確認します。管理対象ドメインを実行している場合は、ホストコントローラーは自動的に名前が一意であることを検証します。

## 8.11. クラスター環境の EJB トランザクション

クライアントコードがクラスター化された EJB を呼び出すと、クラスターのアフィニティーは自動的に設定されます。クライアント側のトランザクションを管理する場合は、クラスター内の特定のノードを対象にする ことを選択することや、クライアントがトランザクションの処理にクラスターノードをレイジーに選択すること もできます。本セクションでは、両方のオプションを説明します。

### 特定のノードの EJB トランザクションを対象にする

以下の手順を使用して、クラスター内の特定のノードを対象にし、トランザクションを処理できます。

1. **InitialContext** の作成時に **PROVIDER_URL** プロパティーを使用してターゲットクラスターノードアドレスを指定します。

```
props.put(Context.PROVIDER_URL, "remote+http://127.0.0.1:8080");
...
InitialContext ctx = new InitialContext(props);
```

2. クライアントで、**InitialContext** から **txn:RemoteUserTransaction** を検索します。

```
UserTransaction ut = (UserTransaction)ctx.lookup("txn:RemoteUserTransaction");
```

**UserTransaction** の Java Naming and Directory Interface ルックアップは、以下のコード例のように **PROVIDER_URL** プロパティーをサーバーの URL に設定し、**txn:UserTransaction** を検索して実行できます。

```
final Hashtable<String, String> jndiProperties = new Hashtable<>();
jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
jndiProperties.put(Context.PROVIDER_URL, "remote+http://localhost:8080");
final Context context = new InitialContext(jndiProperties);

SecuredEJBRemote reference = (SecuredEJBRemote)
context.lookup("txn:UserTransaction");
```

**UserTransaction** は実際の呼び出しが実行されるまで特定の宛先にバインドされません。呼び出し時に、この **UserTransaction** はトランザクションのライフタイム期間中、適切な宛先にバインドされます。

**UserTransaction** を開始する前に、ノード名または宛先を認識する必要はありません。**org.jboss.ejb.client.EJBClient.getUserTransaction ()** メソッドは、最初の呼び出しに基づいて宛先を自動的に選択するリモート **UserTransaction** を提供します。Java Naming and Directory Interface からリモート **UserTransaction** を検索すると、同じ方法で機能します。

**注記**

**org.jboss.ejb.client.EJBClient.getUserTransaction()** メソッドが非推奨になりました。

3. トランザクションが開始されると、すべての EJB 呼び出しがその特定のノードにトランザクションの間バインドされ、サーバーのアフィニティーが確立されます。

4. トランザクションが終了すると、サーバーのアフィニティーがリリースされ、EJB プロキシーが一般的なクラスターアフィニティーに戻ります。

### EJB トランザクションレイジーなノードの選択

トランザクションに関連する最初の呼び出し中に、クライアントがクラスターノードを選択してトランザクションを処理するようにできます。これにより、クラスター全体でトランザクションの負荷分散が可能になります。このオプションを使用するには、以下の手順を行います。

1. EJB を呼び出すために使用される **InitialContext** では **PROVIDER_URL** プロパティーを指定しないでください。
2. クライアントで、**InitialContext** から **txn:RemoteUserTransaction** を検索します。

```
UserTransaction ut = (UserTransaction)ctx.lookup("txn:RemoteUserTransaction");
```

3. トランザクションが開始されると、あるクラスターノードが自動的に選択され、サーバーのアフィニティーが確立されて、すべての EJB 呼び出しがトランザクション期間中その特定のノードにバインドされます。
4. トランザクションが終了すると、サーバーのアフィニティーがリリースされ、EJB プロキシーが一般的なクラスターアフィニティーに戻ります。

# 第9章 EJB 3 サブシステムの調整

**ejb3** サブシステムのパフォーマンスを最適化するための情報は、パフォーマンスチューニングガイドの**EJB サブシステムの調整**を参照してください。

# 付録A リファレンス資料

## A.1. EJB JAVA NAMING AND DIRECTORY INTERFACE リファレンス

セッション bean の Java Naming and Directory Interface ルックアップ名には、以下の構文を使用します。

```
ejb:<appName>/<moduleName>/<distinctName>/<beanName>!<viewClassName>?stateful
```

- **<appName>**: セッション Bean の JAR ファイルがエンタープライズアーカイブ (EAR) 内にデプロイされた場合、**appName** はそれぞれの EAR の名前になります。デフォルトでは、EAR の名前は **.ear** 接尾辞を含まないファイル名です。アプリケーション名は **application.xml** ファイルで上書きできます。セッション Bean が EAR にデプロイされていない場合は、**appName** を空白のままにします。
- **<moduleName>**: **moduleName** は、セッション Bean がデプロイされる JAR ファイルの名前です。JAR ファイルのデフォルト名は、**.jar** 接尾辞を含まないファイル名です。モジュール名は JAR の **ejb-jar.xml** ファイルで上書きできます。
- **<distinctName>**: JBoss EAP では、各デプロイメントで任意の一意の名前を指定できます。デプロイメントに一意の名前がない場合は、**specificName** を空白のままにします。
- **<beanName>**: **beanName** は、呼び出されるセッション Bean の単純なクラス名です。
- **<viewClassName>**: **viewClassName** は、リモートインターフェイスの完全修飾クラス名です。これには、インターフェイスのパッケージ名が含まれます。
- **?stateful**: Java Naming and Directory Interface 名がステートフルセッション Bean を参照する場合は、**?stateful** 接尾辞が必要になります。他の Bean タイプには含まれません。

たとえば、リモートインターフェイス **org.jboss.example.Hello** を公開したステートフル Bean **org.jboss.example.HelloBean** を持つ **hello.jar** をデプロイした場合、Java Naming and Directory Interface ルックアップ名は以下のようになります。

```
ejb:/hello/HelloBean!org.jboss.example.Hello?stateful"
```

## A.2. EJB リファレンス解決

このセクションでは、JBoss EAP による **@EJB** および **@Resource** の実装方法について説明します。XML は常にアノテーションを上書きしますが、同じルールが適用されることに注意してください。

**@EJB アノテーションのルール**

- **@EJB** アノテーションには **mappedName()** 属性もあります。この仕様はこれをベンダー固有のメタデータとして残しますが、JBoss EAP は参照している EJB のグローバル Java Naming and Directory Interface 名として **mappedName()** を認識します。**mappedName()** を指定した場合、他の属性はすべて無視され、バインディングにこのグローバル Java Naming および Directory Interface 名が使用されます。
- 属性が定義されていない状態で **@EJB** を指定する場合:

```
@EJB
ProcessPayment myEjbref;
```

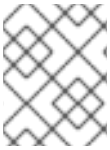
次に、以下のルールが適用されます。

- 参照 Bean の EJB JAR は、**@EJB** インジェクションで使用されるインターフェイスで EJB を検索します。同じビジネスインターフェイスを公開する EJB が複数ある場合、例外が発生します。このインターフェイスを持つ BIOSean が 1 つしかない場合は、その Bean が使用されます。
- EAR でインターフェイスを公開する EJB を検索します。重複が生じると、例外が発生します。それ以外の場合は、一致する Bean を返します。
- JBoss EAP ランタイムで、そのインターフェイスの EJB をグローバルに検索します。同様に、複製が見つかると、例外が発生します。

- **@EJB.beanName()** は **<ejb-link>** に対応します。**beanName()** が定義されている場合、検索で **beanName()** をキーとして使用する以外に定義されていない属性を持つ **@EJB** と同じアルゴリズムを使用します。このルールの例外は、ejb-link **#** 構文を使用する場合です。これは、参照している EJB が置かれている EAR の JAR に相対パスを配置することができます。詳細は EJB 3.2 仕様を参照してください。

## A.3. リモート EJB クライアントのプロジェクト依存関係

リモートクライアントからのセッション Bean の呼び出しが含まれる Maven プロジェクトでは、JBoss EAP Maven リポジトリーから以下の依存関係が必要になります。以下のサブセクションで説明されているように、EJB クライアントの依存関係を宣言する方法があります。



**注記**

**artifactId** バージョンが変更される可能性があります。最新バージョンは、JBoss EAP Maven リポジトリーを参照してください。

### リモート EJB クライアントの Maven 依存関係

**jboss-eap-jakartaee8** Bill of Materials (BOM) は、JBoss EAP アプリケーションで通常必要とされる多くのアーティファクトの正しいバージョンをパッケージ化します。BOM 依存関係は、**import** の範囲で **pom.xml** の **<dependencyManagement>** セクションで指定されています。

**例: POM ファイル <dependencyManagement> セクション**

```
<dependencyManagement>
  <dependencies>
    <dependency>
    <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-jakartaee8</artifactId>
      <version>7.3.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

残りの依存関係は、**pom.xml** ファイルの **<dependencies>** セクションで指定されます。

**例: POM ファイル <dependencies> セクション**

```
<dependencies>
    <!-- Include the Enterprise Java Bean client JARs -->
    <dependency>
        <groupId>org.jboss.eap</groupId>
        <artifactId>wildfly-ejb-client-bom</artifactId>
        <type>pom</type>
    </dependency>

    <!-- Include any additional dependencies required by the application
        ...
    -->

</dependencies>
```

JBoss EAP に同梱される **ejb-remote** クイックスタートは、リモート EJB クライアントアプリケーションの完全な動作例が含まれます。リモートセッション Bean 呼び出しの依存関係設定の完全な例については、クイックスタートのルートディレクトリーにある **client/pom.xml** ファイルを参照してください。

### jboss-ejb-client 依存関係の単一のアーティファクト ID

**wildfly-ejb-client-bom artifactID** を使用し、**jboss-ejb-client** ライブラリーを追加して EJB クライアントに必要な依存関係をすべて含めることができます。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jboss.eap</groupId>
            <artifactId>wildfly-ejb-client-bom</artifactId>
            <version>EJB_CLIENT_BOM_VERSION</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.jboss</groupId>
        <artifactId>jboss-ejb-client</artifactId>
    </dependency>
</dependencies>
```

JBoss EAP Maven リポジトリーで利用可能な**EJB_CLIENT_BOM_VERSION** を使用する必要があります。

## A.4. JBOSS-EJB3.XML デプロイメント記述子の参照

**jboss-ejb3.xml** は、EJB JAR または WAR アーカイブで使用できるカスタムデプロイメント記述子です。EJB JAR アーカイブでは、**META-INF/** ディレクトリーに存在する必要があります。WAR アーカイブでは、**WEB-INF/** ディレクトリーに配置する必要があります。

この形式は **ejb-jar.xml** と同様で、同じネームスペースの一部を使用して追加の名前空間を提供します。**jboss-ejb3.xml** のコンテンツは **ejb-jar.xml** の内容でマージされ、**jboss-ejb3.xml** の内容が優先されます。

本書では、**jboss-ejb3.xml** で使用される追加の非標準の名前空間のみについて説明します。標準の名前空間は、http://java.sun.com/xml/ns/javaee/ を参照してください。

ルート名前空間は http://www.jboss.com/xml/ns/javaee です。

**アセンブリー記述子の名前空間**

以下の名前空間はすべて、**<assembly-descriptor>** 要素で使用できます。これらは、単一の Bean や、ワイルドカード (*) を **ejb-name** として使用してデプロイメントのすべての Bean に設定を適用するために使用できます。

**セキュリティー名前空間 (urn:security)**

```
xmlns:s="urn:security"
```

これにより、EJB の **security-domain** および **run-as-principal** を設定できます。

```
<s:security>
  <ejb-name>*</ejb-name>
  <s:security-domain>myDomain</s:security-domain>
  <s:run-as-principal>myPrincipal</s:run-as-principal>
</s:security>
```

**リソースアダプター名前空間: urn:resource-adapter-binding**

```
xmlns:r="urn:resource-adapter-binding"
```

これにより、メッセージ駆動 Bean のリソースアダプターを設定できます。

```
<r:resource-adapter-binding>
  <ejb-name>*</ejb-name>
  <r:resource-adapter-name>myResourceAdapter</r:resource-adapter-name>
</r:resource-adapter-binding>
```

**IIOP 名前空間: urn:iiop**

```
xmlns:u="urn:iiop"
```

IIOP 名前空間は IIOP 設定が設定される場所です。

**プール名前空間: urn:ejb-pool:1.0**

```
xmlns:p="urn:ejb-pool:1.0"
```

これにより、含まれるステートレスセッション Bean またはメッセージ駆動 Bean が使用するプールを選択できます。プールはサーバー設定で定義されます。

```
<p:pool>
  <ejb-name>*</ejb-name>
  <p:bean-instance-pool-ref>my-pool</p:bean-instance-pool-ref>
</p:pool>
```

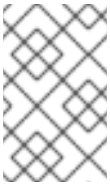
**キャッシュ名前空間: urn:ejb-cache:1.0**

```
xmlns:c="urn:ejb-cache:1.0"
```

これにより、含まれるステートフルセッション Bean によって使用されるキャッシュを選択できます。キャッシュはサーバー設定で定義されます。

```
<c:cache>
  <ejb-name>*</ejb-name>
  <c:cache-ref>my-cache</c:cache-ref>
</c:cache>
```

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
    xmlns="http://java.sun.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-ejb3-2_0.xsd"
    version="3.1"
    impl-version="2.0">
  <enterprise-beans>
    <message-driven>
      <ejb-name>ReplyingMDB</ejb-name>
      <ejb-class>org.jboss.as.test.integration.ejb.mdb.messagedestination.ReplyingMDB</ejb-
class>
      <activation-config>
        <activation-config-property>
          <activation-config-property-name>destination</activation-config-property-name>
          <activation-config-property-value>java:jboss/mdbtest/messageDestinationQueue
          </activation-config-property-value>
        </activation-config-property>
      </activation-config>
    </message-driven>
  </enterprise-beans>
 </jboss:ejb-jar>
```



**注記**

**jboss-ejb3-spec-2_0.xsd** ファイルには、スキーマバリデーションエラーが発生する可能性がある既知の問題があります。これらのエラーは無視して構いません。詳細は、https://bugzilla.redhat.com/show_bug.cgi?id=1192591 を参照してください。

## A.5. EJB スレッドプールの設定

EJB スレッドプールは、管理コンソールまたは管理 CLI を使用して作成できます。

### A.5.1. 管理コンソールを使用した EJB スレッドプールの設定

**手順**

1. 管理コンソールにログインします。
2. **Configuration → Subsystems → EJB** の順に移動し、**View** をクリックします。

3. **Container → Thread Pool** の順に選択します。
4. **Add** をクリックして、**Name** および **Max Threads** 値を指定します。
5. **Save** をクリックします。

### A.5.2. 管理 CLI を使用した EJB スレッドプールの設定

**手順**

1. 以下の構文で **add** 操作を使用します。

```
/subsystem=ejb3/thread-pool=THREAD_POOL_NAME:add(max-threads=MAX_SIZE)
```

   a. **THREAD_POOL_NAME** は、スレッドプールに必要な名前に置き換えます。

   b. **MAX_SIZE** はスレッドプールの最大サイズに置き換えます。

2. **read-resource** 操作を使用して、スレッドプールの作成を確認します。

```
/subsystem=ejb3/thread-pool=THREAD_POOL_NAME:read-resource
```

   a. **ejb3** サブシステムのすべてのサービスを、新しいスレッドプールを使用するように再設定するには、以下のコマンドを使用します。

```
/subsystem=ejb3/thread-pool=bigger:add(max-threads=100, core-threads=10)
/subsystem=ejb3/service=async:write-attribute(name=thread-pool-name, value="bigger")
/subsystem=ejb3/service=remote:write-attribute(name=thread-pool-name,
value="bigger")
/subsystem=ejb3/service=timer-service:write-attribute(name=thread-pool-name,
value="bigger")
reload
```

**XML 設定例:**

```
<subsystem xmlns="urn:jboss:domain:ejb3:5.0">
    ...
    <async thread-pool-name="bigger"/>
    ...
    <timer-service thread-pool-name="bigger" default-data-store="default-file-store">
    ...
    <remote connector-ref="http-remoting-connector" thread-pool-name="bigger"/>
    ...
    <thread-pools>
        <thread-pool name="default">
            <max-threads count="10"/>
            <core-threads count="5"/>
            <keepalive-time time="100" unit="milliseconds"/>
        </thread-pool>
        <thread-pool name="bigger">
            <max-threads count="100"/>
            <core-threads count="5"/>
```

```
    </thread-pool>
</thread-pools>
...
```

### A.5.3. EJB スレッドプールの属性

特定の設定のニーズに応じてより効率的に実行するには、属性を使用して EJB スレッドプールを設定できます。

- **max-threads** 属性は、エグゼキューターがサポートするスレッドの合計数または最大数を決定します。

```
/subsystem=ejb3/thread-pool=default:write-attribute(name=max-threads, value=9)
{"outcome" => "success"}
```

- **core-threads** 属性は、エグゼキューターのプールに保持されるスレッドの数を決定します。これには、アイドル状態のスレッドが含まれます。**core-threads** 属性が指定されていない場合、デフォルトは **max-threads** の値に設定されます。

```
/subsystem=ejb3/thread-pool=default:write-attribute(name=core-threads, value=3)
{"outcome" => "success"}
```

- **keepalive-time** 属性は、コア以外のスレッドがアイドル状態でいられる時間を決定します。この時間を過ぎると、コア以外のスレッドが削除されます。

```
/subsystem=ejb3/thread-pool=default:write-attribute(name=keepalive-time, value={time=5,
unit=MINUTES})
{"outcome"=> "success"}
```

- **keepalive-time** 属性の時間単位を変更せずに時間を変更するには、以下のコマンドを使用します。

```
/subsystem=ejb3/thread-pool=default:write-attribute(name=keepalive-time.time, value=10)
{"outcome"=> "success"}
```

Revised on 2023-01-28 12:51:12 +1000