JBoss Enterprise Application Platform 6.1

開発ガイド

JBoss Enterprise Application Platform 6 向け

エディッション 3

Last Updated: 2017-11-01

JBoss Enterprise Application Platform 6.1 開発ガイド

JBoss Enterprise Application Platform 6 向け
エディッション 3

Nidhi Chaudhary

Lucas Costi

Russell Dickenson

Sande Gilda

Vikram Goyal

Eamon Logue

Darrin Mison

Scott Mumford

David Ryan

Misty Stanley-Jones

Keerat Verma

Tom Wells

法律上の通知

概要

本書は、JBoss Enterprise Application Platform 6 とそのパッチリリースを使用する Java EE 6 の開発者向けの参考資料や例を提供します。

目次

第1章 アプリケーションの開発

1.1. はじめに

1.1.1. JBoss Enterprise Application Platform 6 について

JBoss Enterprise Application Platform 6 は高速でセキュアな高性能ミドルウェアプラットフォームで、オープンな標準に基づいて構築され、Java Enterprise Edition 6 の仕様に準拠しています。高可用性クラスタリング、強力なメッセージング、分散キャッシングなどの技術を JBoss Application Server 7 と統合し、安定したスケーラブルな高速プラットフォームを作り上げます。

新しいモジュラー構造により、必要な時だけサービスを有効にできるため、起動速度が大幅に向上します。管理コンソールと管理コマンドラインインターフェースを使用すると、XML 設定ファイルを手作業で編集する必要がなくなるため、スクリプトを作成して作業を自動化することが可能です。さらに、API と開発フレームワークも含まれており、これらを使用して堅牢で拡張性のある、セキュアな Java EE アプリケーションを迅速に開発することができます。

バグを報告する

1.2. 要件

1.2.1. Java Enterprise Edition 6 を理解する

1.2.1.1. EE 6 プロファイルの概要

Java Enterprise Edition 6 (EE 6) には、複数のプロファイルのサポート (つまり、API のサブセット) が含まれます。EE 6 の仕様で定義されるプロファイルは、*Full Profile* と *Web Profile* の 2 つだけです。

EE 6 Full Profile には、EE 6 の仕様に含まれるすべての API と仕様が含まれます。EE 6 の Web Profile には、Web 開発者にとって有用な API のサブセットが含まれます。

JBoss Enterprise Application Platform 6 は、Java Enterprise Edition 6 の Full Profile および Web Profile の仕様の認定された実装です。

- 「Java Enterprise Edition 6 Web Profile」
- 「Java Enterprise Edition 6 Full Profile」

バグを報告する

1.2.1.2. Java Enterprise Edition 6 Web Profile

Web Profileは、Java Enterprise Edition 6 仕様で定義されている 2 つのプロファイルのうちの１つです。Web Profile は Web アプリケーション開発向けに設計されており、Java Enterprise Edition 6 仕様で定義されている、もう１つのプロファイルは Full Profile です。詳細については、「Java Enterprise Edition 6 Full Profile」を参照してください。

Java EE 6 Web Profile の要件

- Java Platform、Enterprise Edition 6
- Java Web テクノロジー

- Servlet 3.0 (JSR 315)
- JSP 2.2 および Expression Language (EL) 1.2
- JavaServer Faces (JSF) 2.0 (JSR 314)
- JSP 1.2 向けの Java Standard Tag Library (JSTL)
- Debugging Support for Other Languages 1.0 (JSR 45)

- エンタープライズアプリケーションテクノロジー
  - Contexts and Dependency Injection (CDI) (JSR 299)
  - Dependency Injection for Java (JSR 330)
  - Enterprise JavaBeans 3.1 Lite (JSR 318)
  - Java Persistence API 2.0 (JSR 317)
  - Common Annotations for the Java Platform 1.1 (JSR 250)
  - Java Transaction API (JTA) 1.1 (JSR 907)
  - Bean Validation (JSR 303)

バグを報告する

1.2.1.3. Java Enterprise Edition 6 Full Profile

Java Enterprise Edition 6 (EE 6) の仕様では、プロファイルのコンセプトを定義し、仕様の一部として 2 つのプロファイルを定義しています。Java Enterprise Edition 6 Web Profile (「Java Enterprise Edition 6 Web Profile」) でサポートされているアイテム以外に、Full Profile では以下の API がサポートされます。JBoss Enterprise Edition 6 は Full Profile をサポートします。

EE 6 Full Profile に含まれるアイテム

- EJB 3.1 (Lite ではない) (JSR 318)
- Java EE Connector Architecture 1.6 (JSR 322)
- Java Message Service (JMS) API 1.1 (JSR 914)
- JavaMail 1.4 (JSR 919)

- **Web サービステクノロジー**
  - Jax-RS RESTful Web Services 1.1 (JSR 311)
  - Implementing Enterprise Web Services 1.3 (JSR 109)
  - JAX-WS Java API for XML-Based Web Services 2.2 (JSR 224)
  - Java Architecture for XML Binding (JAXB) 2.2 (JSR 222)
  - Web Services Metadata for the Java Platform (JSR 181)

- Java APIs for XML-based RPC 1.1 (JSR 101)
- Java APIs for XML Messaging 1.3 (JSR 67)
- Java API for XML Registries (JAXR) 1.0 (JSR 93)

- 管理およびセキュリティテクノロジー
  - Java Authentication Service Provider Interface for Containers 1.0 (JSR 196)
  - Java Authentication Contract for Containers 1.3 (JSR 115)
  - Java EE Application Deployment 1.2 (JSR 88)
  - J2EE Management 1.1 (JSR 77)

バグを報告する

1.2.2. JBoss Enterprise Application Platform 6 で使用されるモジュールと新しいモジュラークラスローディングシステムについて

1.2.2.1. モジュール

モジュールは、クラスのロードと依存関係の管理に使用される、クラスの論理的なグループです。JBoss Enterprise Application Platform 6 では、静的モジュールと動的モジュールと呼ばれる 2 種類のモジュールが存在します。ただし、この 2 種類のモジュールの違いは、パッケージ化の方法のみです。すべてのモジュールは同じ機能を提供します。

静的モジュール

静的モジュールは、アプリケーションサーバーの ***EAP_HOME*/modules/** ディレクトリに事前定義されます。各サブディレクトリは 1 つのモジュールを表し、1 つまたは複数の JAR ファイルと設定ファイル (**module.xml**) が含まれます。モジュールの名前は、**module.xml** ファイルで定義されます。アプリケーションサーバーで提供されるすべての API (Java EE API や JBoss Logging などの他の API を含む) は、静的モジュールとして提供されます。

カスタム静的モジュールの作成は、同じサードパーティーライブラリを使用する同じサーバー上に多くのアプリケーションがデプロイされる場合に役立ちます。これらのライブラリを各アプリケーションとバンドルする代わりに、JBoss 管理者はこれらのライブラリが含まれるモジュールを作成およびインストールできます。アプリケーションは、カスタム静的モジュールで明示的な依存関係を宣言できます。

動的モジュール

動的モジュールは、各 JAR または WAR デプロイメント (または、EAR 内のサブデプロイメント) に対してアプリケーションサーバーによって作成およびロードされます。動的モジュールの名前は、デプロイされたアーカイブの名前から派生されます。デプロイメントはモジュールとしてロードされるため、依存関係を設定でき、他のデプロイメントが依存関係として使用できます。

モジュールは必要なときのみロードされます。通常、明示的または暗黙的な依存関係があるアプリケーションがデプロイされるときのみ、モジュールがロードされます。

バグを報告する

1.2.2.2. クラスロードとモジュールの概要

JBoss Enterprise Application Platform 6 は、デプロイされたアプリケーションのクラスパスを制御するために新しいモジュール形式のクラスロードシステムを使用します。このシステムでは、階層クラスローダーの従来のシステムよりも、柔軟性と制御が強化されています。開発者は、アプリケーションで利用可能なクラスに対して粒度の細かい制御を行い、アプリケーションサーバーで提供されるクラスを無視して独自のクラスを使用してデプロイメントを設定できます。

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

以下に、開発者が JBoss Enterprise Application Platform 6 でアプリケーションを正しくビルドおよびデプロイするために知る必要があることを示します。

バグを報告する

1.3. 開発環境の設定

1.3.1. JBoss Developer Studio のダウンロードとインストール

1.3.1.1. JBoss Developer Studio の設定

1. 「JBoss Developer Studio 5 のダウンロード」
2. 「JBoss Developer Studio 5 のインストール」
3. 「JBoss Developer Studio の起動」

バグを報告する

1.3.1.2. JBoss Developer Studio 5 のダウンロード

1. https://access.redhat.com/ にアクセスします。
2. **ダウンロード** → **JBoss Enterprise Middleware** → **ダウンロード** と選択します。
3. ドロップボックスから **JBoss Developer Studio** を選択します。
4. 適切なバージョンを選択し、**ダウンロード** をクリックします。

バグを報告する

1.3.1.3. JBoss Developer Studio 5 のインストール

前提条件

「JBoss Developer Studio 5 のダウンロード」

手順1.1 JBoss Developer Studio 5 のインストール

1. 端末を開きます。
2. ダウンロードした **.jar** ファイルが含まれるディレクトリへ移動します。

3. 次のコマンドを実行して GUI インストーラーを開始します。

```
java -jar jbdevstudio-build_version.jar
```

4. **[Next]** をクリックしてインストールを開始します。
5. **[I accept the terms of this license agreement]** を選択し、**[Next]** をクリックします。
6. インストールパスを調整し、**[Next]** をクリックします。

注記

インストールパスのフォルダーが存在しない場合はメッセージが表示されます。**[Ok]** をクリックしてフォルダーを作成します。

7. デフォルトの JVM が選択されます。他の JVM を選択するか、そのまま **[Next]** をクリックします。
8. 使用可能なアプリケーションプラットフォームを追加し、**[Next]** をクリックします。
9. インストールの詳細を確認し、**[Next]** をクリックします。
10. インストールが完了したら **[Next]** をクリックします。
11. JBoss Developer Studio のデスクトップショートカットを設定し、**[Next]** をクリックします。
12. **[Done]** をクリックします。

バグを報告する

1.3.1.4. JBoss Developer Studio の起動

前提条件

「JBoss Developer Studio 5 のインストール」

手順1.2 JBoss Developer Studio を起動するコマンド

1. 端末を開きます。
2. インストールディレクトリへ移動します。
3. 次のコマンドを実行して JBoss Developer Studio を起動します。

```
[localhost]$ ./jbdevstudio
```

バグを報告する

1.3.1.5. JBoss Enterprise Application Platform 6 サーバーの JBoss Developer Studio への追加

次の手順では、JBoss Developer Studio を初めて使用し、JBoss Enterprise Application Platform 6 サーバーを追加したことがないことを前提としています。

手順1.3 サーバーの追加

1. **[Servers]**タブを開きます。**[Servers]**タブがない場合は次のようにパネルへ追加します。
   a. **[Window]** → **[Show View]** → **[Other...]** をクリックします。
   b. **[Servers]** フォルダーより **[Server]** を選択し、**[OK]** をクリックします。
2. **[new server wizard]**リンクをクリックするか、空のサーバーパネル内で右クリックし、**[New]** → **[Server]** と選択します。

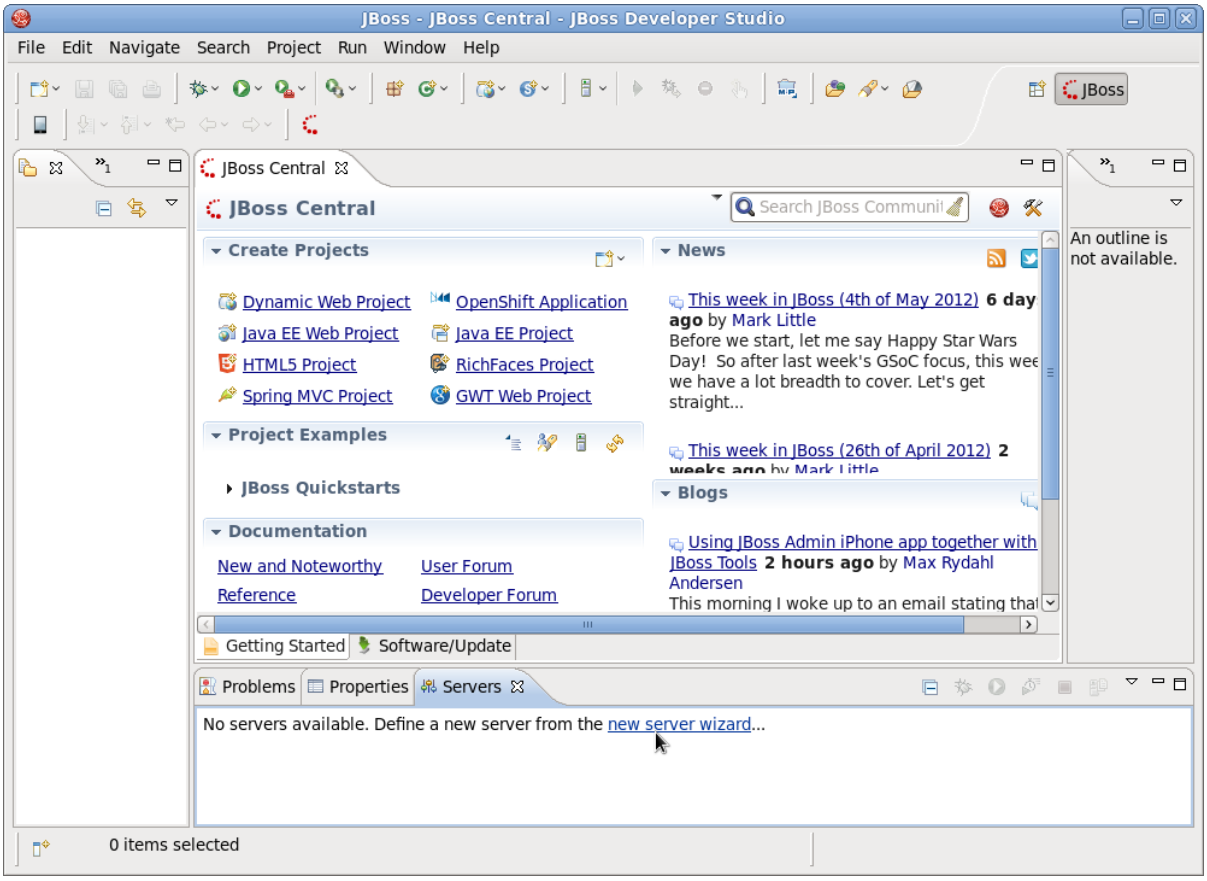


図1.1 新しいサーバーの追加 - 使用できるサーバーがない場合

3. **[JBoss Enterprise Middleware]** を拡張し、 **[JBoss Enterprise Application Platform 6.x]** を選択します。その後、**[Next]** ボタンをクリックします。

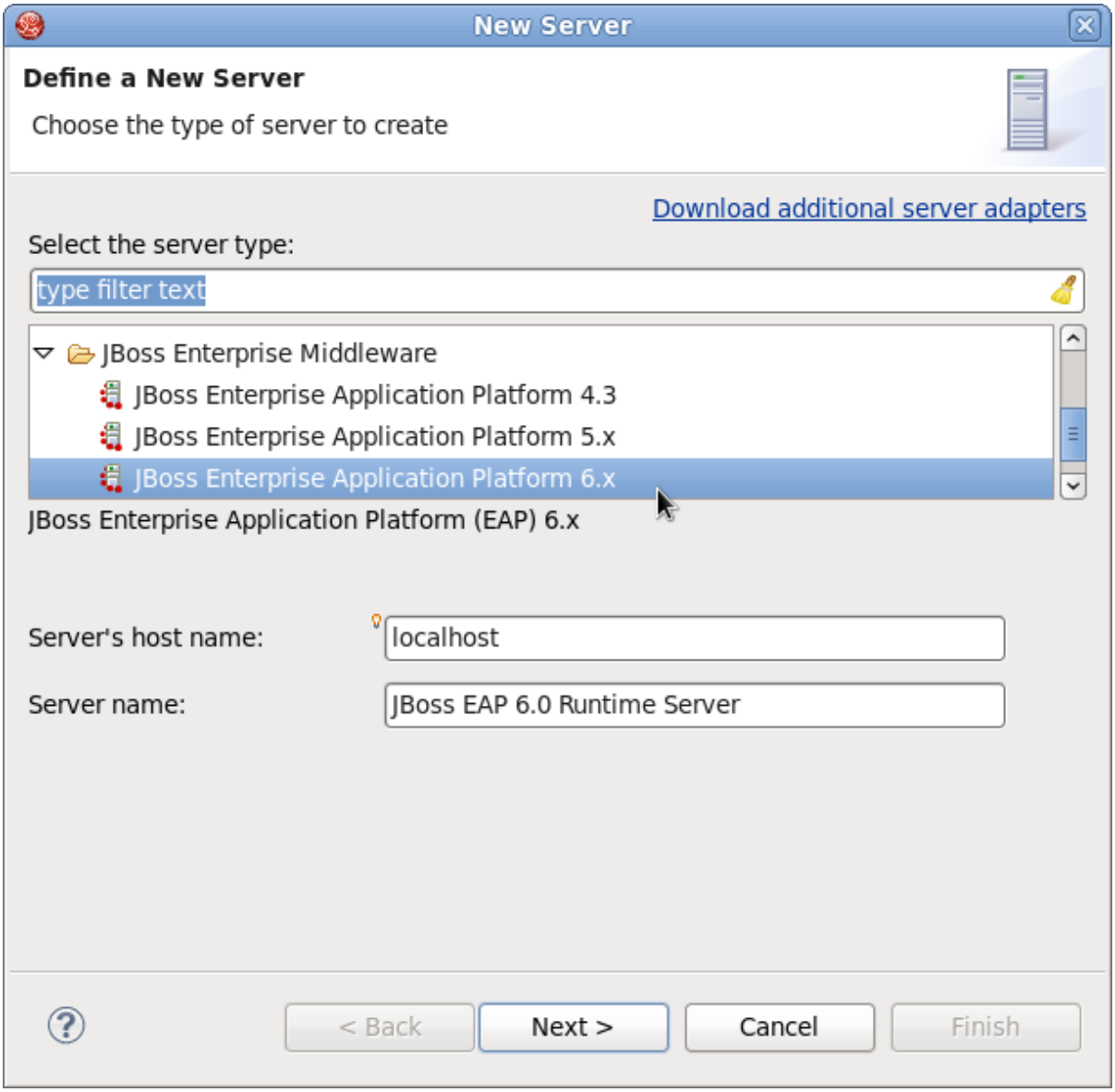


図1.2 サーバータイプの選択

4. **[Browse]** をクリックし、JBoss Enterprise Application Platform 6 のインストール場所へ移動します。そして **[Next]** をクリックします。

図1.3 サーバーインストールの閲覧

5. この画面でサーバーの動作を定義します。手作業でサーバーを起動するか、JBoss Developer Studio に管理を任せます。デプロイメントのリモートサーバーを定義し、そのサーバーの管理ポートを公開するかどうかを決定できます (たとえば、JMX を使用してこのサーバーに接続する必要がある場合)。この例では、サーバーがローカルサーバーであり、JBoss Developer Studio がサーバーを管理するため、何もチェックする必要がないことを前提とします。**次へ (Next)** をクリックします。

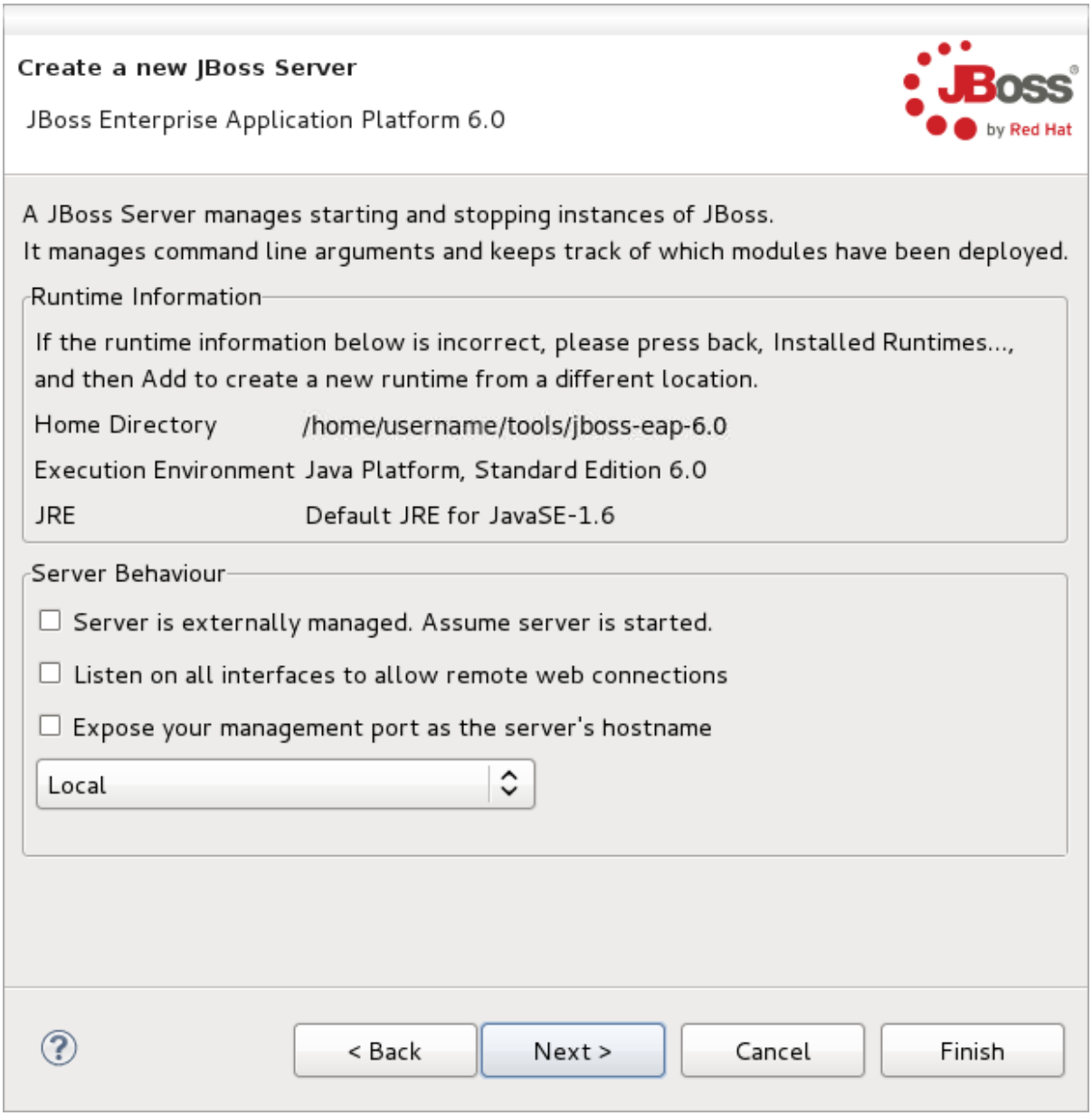


図1.4 新しい JBoss サーバーの挙動の定義

6. この画面により新しいサーバーに対して既存のプロジェクトを設定することが可能です。現時点ではプロジェクトがないため、**[Finish]** をクリックします。

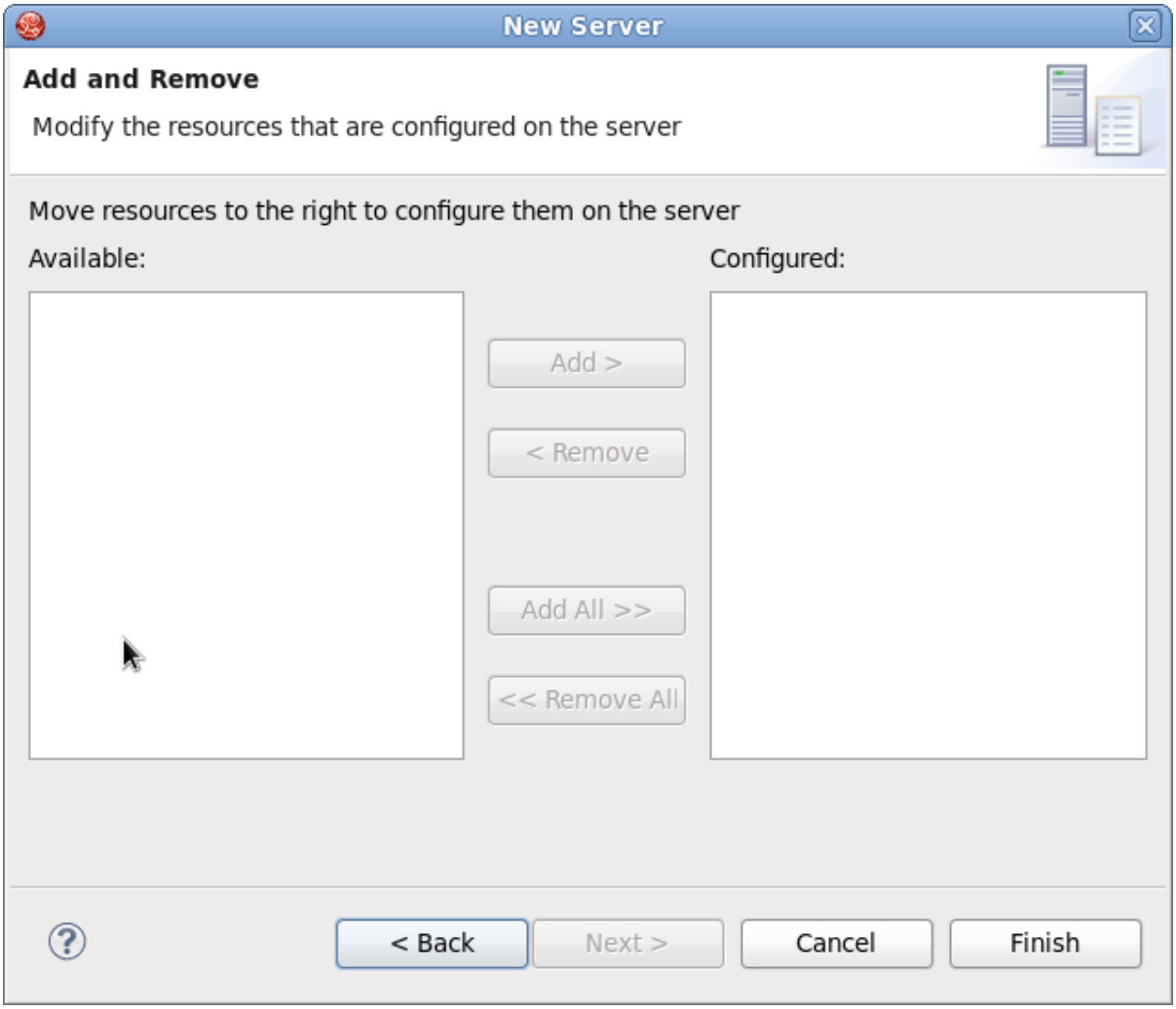


図1.5 新しい JBoss サーバーのリソースの変更

結果

JBoss Enterprise Application Server 6.0 のランタイムサーバーは **[Servers]** タブに表示されます。

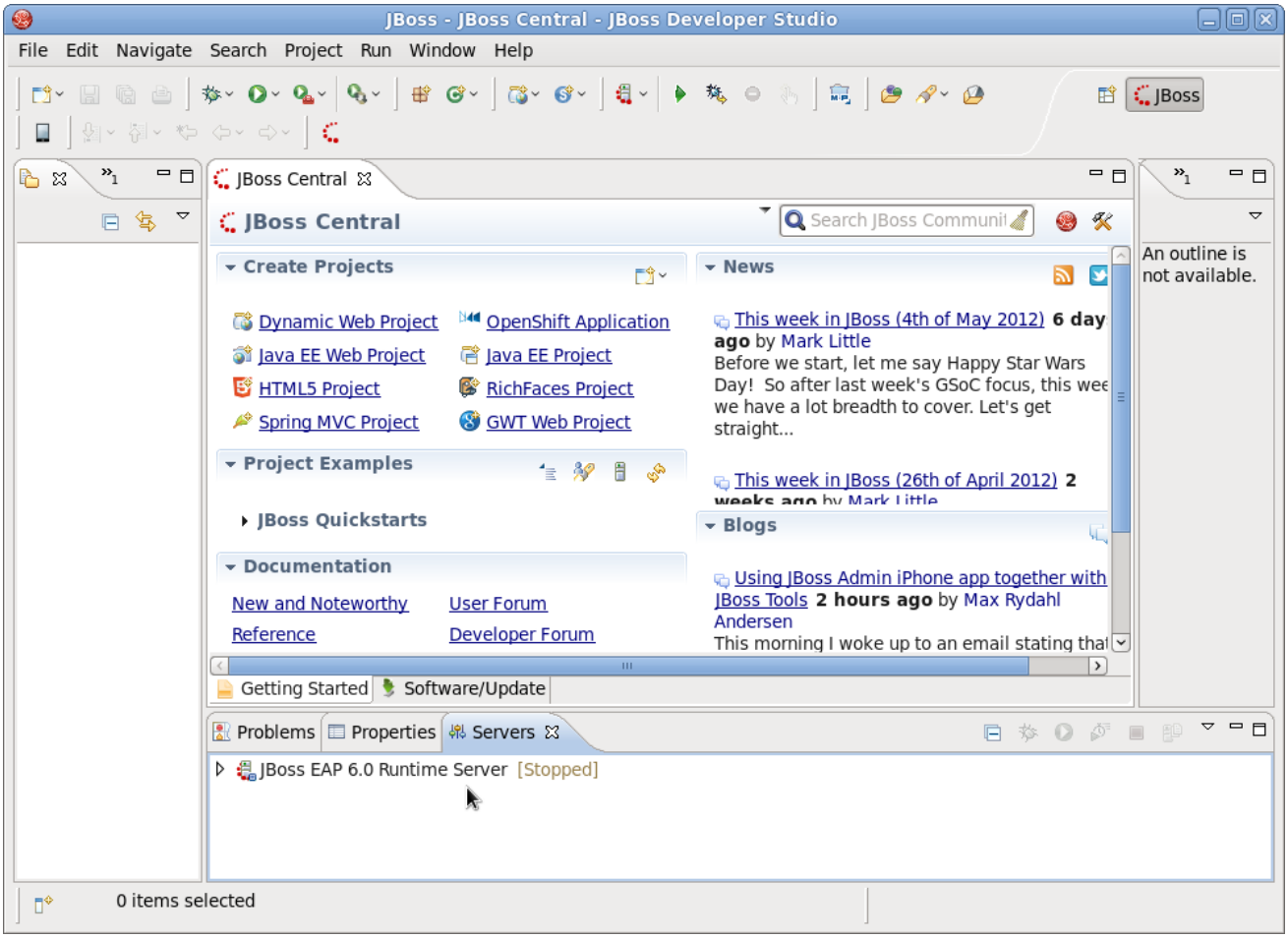


図1.6 サーバーがサーバーリストに表示される

バグを報告する

1.4. 最初のアプリケーションの実行

1.4.1. デフォルトの Welcome Web アプリケーションの置き換え

JBoss Enterprise Application Platform 6 には、8080 番ポートでサーバーの URL を開くと表示される Welcome アプリケーションが含まれています。次の手順でこのアプリケーションを独自の Web アプリケーションに置き換えることができます。

手順1.4 デフォルトの Welcome Web アプリケーションを独自の Web アプリケーションに置き換える

1. **Welcome アプリケーションを無効にします。**
   管理 CLI スクリプト ***EAP_HOME*/bin/jboss-cli.sh** を使用して次のコマンドを実行します。異なる管理対象ドメインプロファイルの変更が必要となる場合があります。スタンドアローンサーバーでは、コマンドの **/profile=default** 部分の削除が必要となる場合があります。

   ```
   /profile=default/subsystem=web/virtual-server=default-host:write-attribute(name=enable-welcome-root,value=false)
   ```

2. **ルートコンテキストを使用するよう Web アプリケーションを設定します。**
   Web アプリケーションを設定してルートコンテキストを (/) を URL アドレスとして使用するには、**META-INF/** または **WEB-INF/** ディレクトリにある **jboss-web.xml** を変更します。**<context-root>** ディレクティブを次のようなディレクティブに置き換えます。

```
<jboss-web>
    <context-root>/</context-root>
</jboss-web>
```

3. **アプリケーションをデプロイします。**
   サーバーグループか最初に変更したサーバーに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 **`http://SERVER_URL:PORT/`** で使用できるようになります。

バグを報告する

1.4.2. クイックスタートコードの例をダウンロードする

1.4.2.1. Java EE クイックスタートサンプルへのアクセス

概要

JBoss Enterprise Application Platform 6 には、ユーザーが Java EE 6 の技術を使用したアプリケーションの作成を簡単に開始できるクイックスタートのサンプルが複数含まれています。

要件

- Maven 3.0.0 以降のバージョン。Maven のインストールに関する詳細は http://maven.apache.org/download.html を参照してください。
- 「Maven リポジトリについて」
- 「JBoss Enterprise Application Platform 6 の Maven リポジトリのローカルインストール」
- 「Maven 設定を使用した JBoss Enterprise Application Platform の Maven リポジトリーの設定」

手順1.5 クイックスタートのダウンロード

1. Web ブラウザーを開き、URL https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=appplatform にアクセスします。
2. リストに「Application Platform 6 クイックスタート」があることを確認します。
3. **[ダウンロード]** ボタンをクリックし、サンプルが含まれる **`.zip`** ファイルをダウンロードします。
4. 希望のディレクトリにアーカイブを展開します。

結果

Java EE クイックスタートのサンプルがダウンロードされ、解凍されます。各クイックスタートのデプロイ方法については、**`jboss-eap-6.0-quickstarts/`** ディレクトリーにある **`README.md`** ファイルを参照してください。

バグを報告する

1.4.3. クイックスタートの実行

1.4.3.1. JBoss Developer Studio でのクイックスタートの実行

手順1.6 JBoss Developer Studio にクイックスタートをインポートする

各クイックスタートには、クイックスタートのプロジェクトおよび設定情報が含まれる POM (プロジェクトオブジェクトモデル) ファイルが同梱されています。この POM ファイルを使用すると、簡単にクイックスタートを JBoss Developer Studio へインポートすることができます。

1. この作業を行っていない場合は、「Maven 設定を使用した JBoss Enterprise Application Platform の Maven リポジトリーの設定」に記載された手順に従ってください。
2. JBoss Developer Studio を起動します。
3. メニューより **[File]** → **[Import]** と選択します。
4. 選択リストより **[Maven]** → **[Maven Projects]** と選択し、**[Next]** を選択します。

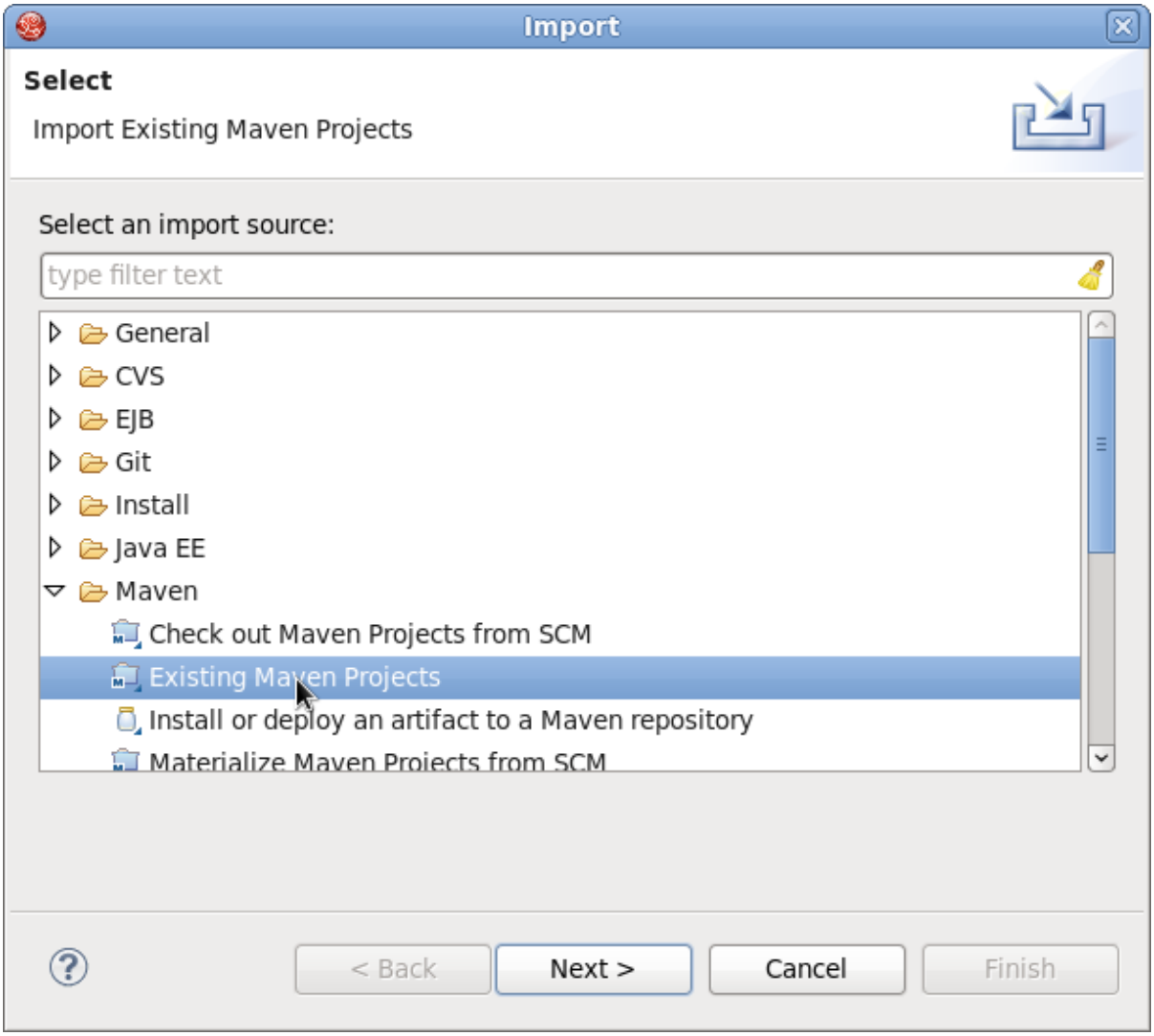


図1.7 既存の Maven プロジェクトのインポート

5. インポートするクイックスタートのディレクトリーを参照し、**OK** をクリックします。**[Projects]** リストボックスに、選択したクイックスタートプロジェクトの**pom.xml** ファイルが示されます。

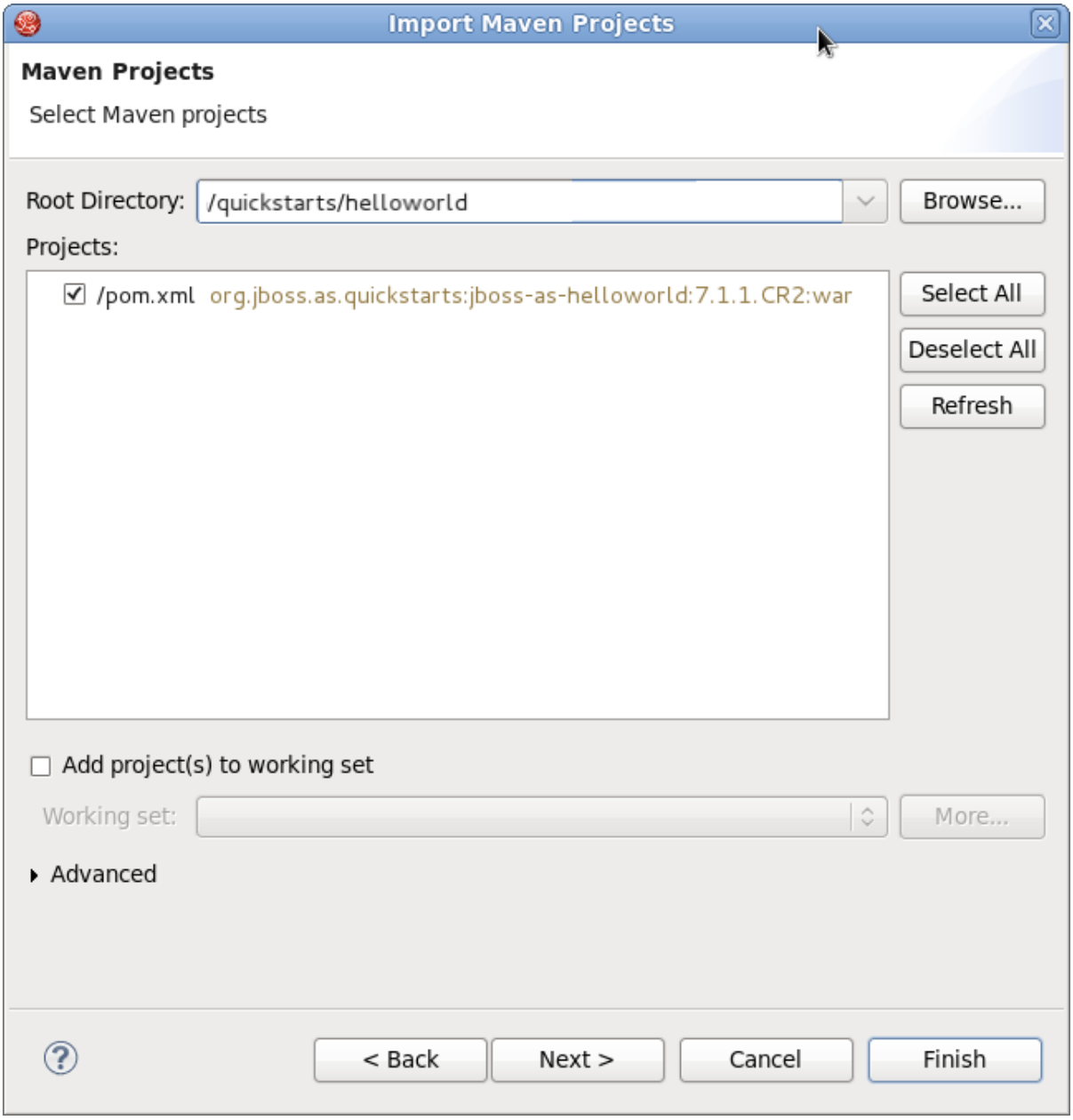


図1.8 Maven プロジェクトの選択

6. **[Next]** をクリックした後、 **[Finish]** をクリックします。

手順1.7 helloworld クイックスタートのビルドとデプロイ

helloworld クイックスタートは最も単純なクイックスタートの 1 つで、JBoss サーバーが適切に設定され実行されているか検証することができます。

1. **[Servers]** タブを開き、パネルにアプリケーションを追加します。
   a. **[Window]** → **[Show View]** → **[Other...]** をクリックします。
   b. **[Servers]** フォルダーから **[Server]** を選択し**[Ok]** をクリックします。
2. **[Project Explorer]** タブで **[helloworld]** を右クリックし、**[Run As]** → **[Run on Server]** を選択します。
3. **[JBoss EAP 6.0 Runtime Server]** サーバーを選択し、**[Next]** をクリックします。これにより **helloworld** クイックスタートが JBoss サーバーにデプロイされます。

4. **helloworld** が JBoss サーバーに正しくデプロイされたことを確認するには、Web ブラウザーを開いて、URL http://localhost:8080/jboss-as-helloworld でアプリケーションに接続します。

バグを報告する

1.4.3.2. コマンドラインを使用したクイックスタートの実行

手順1.8 コマンドラインを使用したクイックスタートのビルドおよびデプロイ

コマンドラインを使用すると簡単にクイックスタートをビルドおよびデプロイすることができます。コマンドラインを使用する場合、JBoss サーバーを起動する必要がある時はユーザーが起動しなければならないため注意してください。

1. **クイックスタートのルートディレクトリにある README ファイルを確認してください。**
   このファイルにはシステム要件に関する一般的な情報、Maven の設定方法、ユーザーの追加方法、クイックスタートの実行方法が含まれています。クイックスタートを始める前に必ず読むようにしてください。

   このファイルには使用可能なクイックスタートの一覧表も含まれています。この表にはクイックスタート名と使用する技術が記載され、各クイックスタートの簡単な説明と設定するために必要な経験レベルが記載されています。クイックスタートの詳細情報はクイックスタート名をクリックしてください。

   他のクイックスタートを向上したり拡張するため作成されたクイックスタートもあります。このようなクイックスタートは **Prerequisites** カラムに記載されています。クイックスタートに前提条件がある場合、クイックスタートを始める前にこれらをインストールする必要があります。

   任意コンポーネントのインストールや設定が必要になるクイックスタートもあります。これらのコンポーネントはクイックスタートが必要である場合のみインストールしてください。

2. **helloworld クイックスタートを実行します。**
   helloworld クイックスタートは最も単純なクイックスタートの 1 つで、JBoss サーバーが適切に設定され実行されているか検証することができます。**helloworld** クイックスタートのルートにある **README** ファイルを開きます。このファイルにはクイックスタートのビルドおよびデプロイ方法や実行しているアプリケーションへのアクセス方法の詳細手順が含まれています。

3. **別のクイックスタートを実行します。**
   各クイックスタートのルートフォルダーにある **README** ファイルの手順に従って例を実行します。

バグを報告する

1.4.4. クイックスタートチュートリアルの確認

1.4.4.1. helloworld クイックスタート

概要

helloworld クイックスタートでは JBoss Enterprise Application Platform 6 に単純なサーブレットをデプロイする方法を説明します。ビジネスロジックは CDI (Contexts and Dependency Injection) Bean として提供されるサービスにカプセル化されサーブレットに挿入されます。このクイックスタートは大

変単純です。「Hello World」を Web ページに出力するだけです。サーバーが適切に設定され、起動されたかを最初に確認するのに適しています。

コマンドラインを使用してこのクイックスタートをビルドしデプロイする手順の詳細は **helloworld** クイックスタートディレクトリのルートにある README ファイルを参照してください。ここでは JBoss Developer Studio を使用してクイックスタートを実行する方法を説明します。

手順1.9 helloworld クイックスタートを JBoss Developer Studio にインポートします。

「JBoss Developer Studio でのクイックスタートの実行」に記述された手順に従ってすでにすべてのクイックスタートを JBoss Developer Studio にインポートした場合は、次のセクションに進むことができます。

1. インポートしていない場合は、「Maven 設定を使用した JBoss Enterprise Application Platform の Maven リポジトリーの設定」に記述された手順に従います。
2. JBoss Developer Studio がインストールされていない場合は、「JBoss Developer Studio 5 のインストール」に記述された手順に従います。
3. 「JBoss Developer Studio の起動」に記述された手順に従います。
4. メニューより **[File]** → **[Import]** と選択します。
5. 選択リストより **[Maven]** → **[Maven Projects]** と選択し、**[Next]** を選択します。

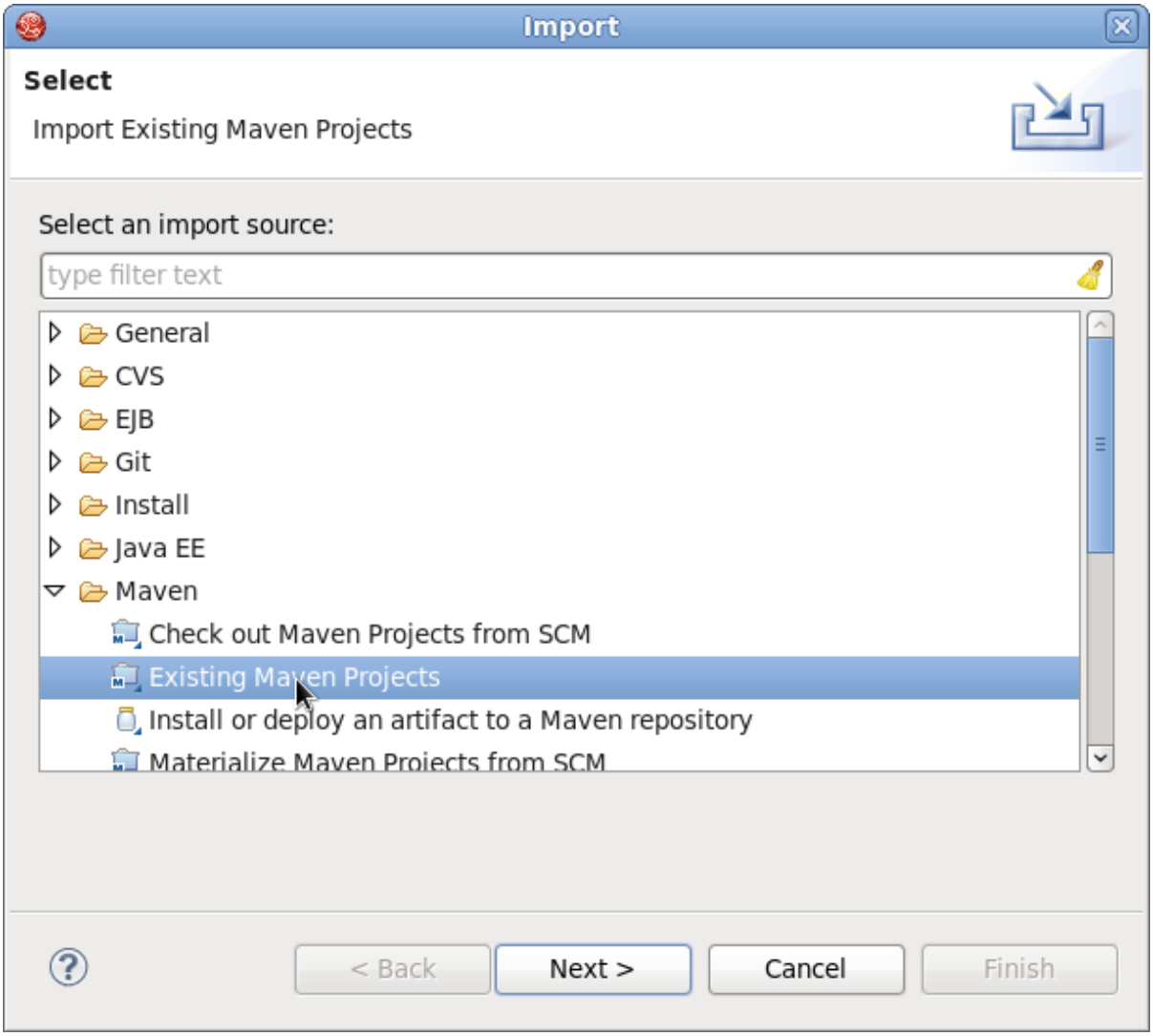


図1.9 既存の Maven プロジェクトのインポート

6. ***QUICKSTART_HOME*/quickstart/helloworld/** ディレクトリを閲覧し、**[OK]** をクリックします。**[Projects]** リストボックスに **[helloworld]** クイックスタートプロジェクトから **pom.xml** ファイルが追加されます。

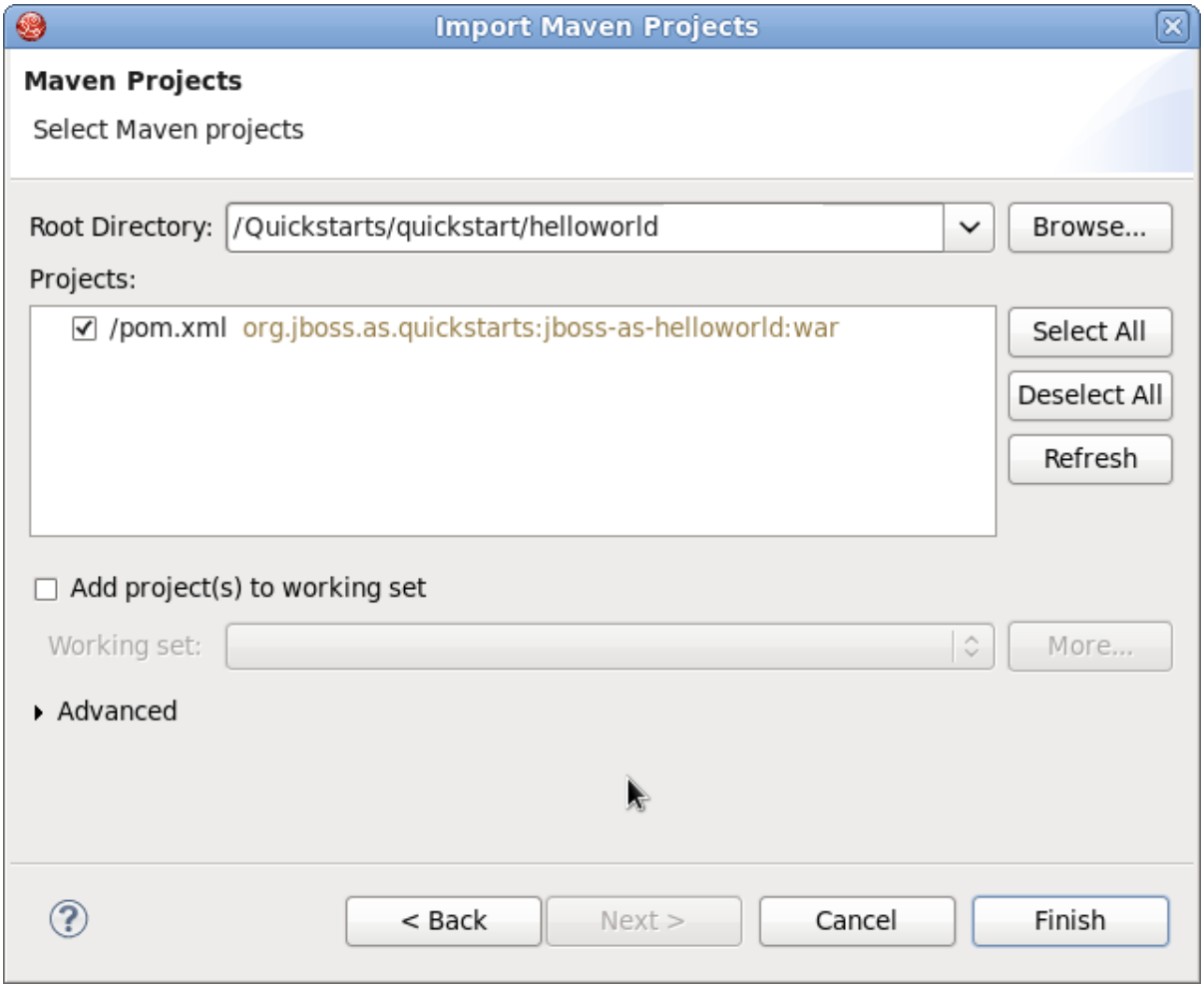


図1.10 Maven プロジェクトの選択

7. **[Finish]** をクリックします。

手順1.10 helloworld クイックスタートのビルドとデプロイ

1. JBoss Enterprise Application Platform 6 用 JBoss Developer Studio がまだ設定されていない場合は、「JBoss Enterprise Application Platform 6 サーバーの JBoss Developer Studio への追加」に記述された手順に従います。

2. **[Project Explorer]** タブの **[jboss-as-helloworld]** を右クリックし、**[Run As]** → **[Run on Server]** と選択します。

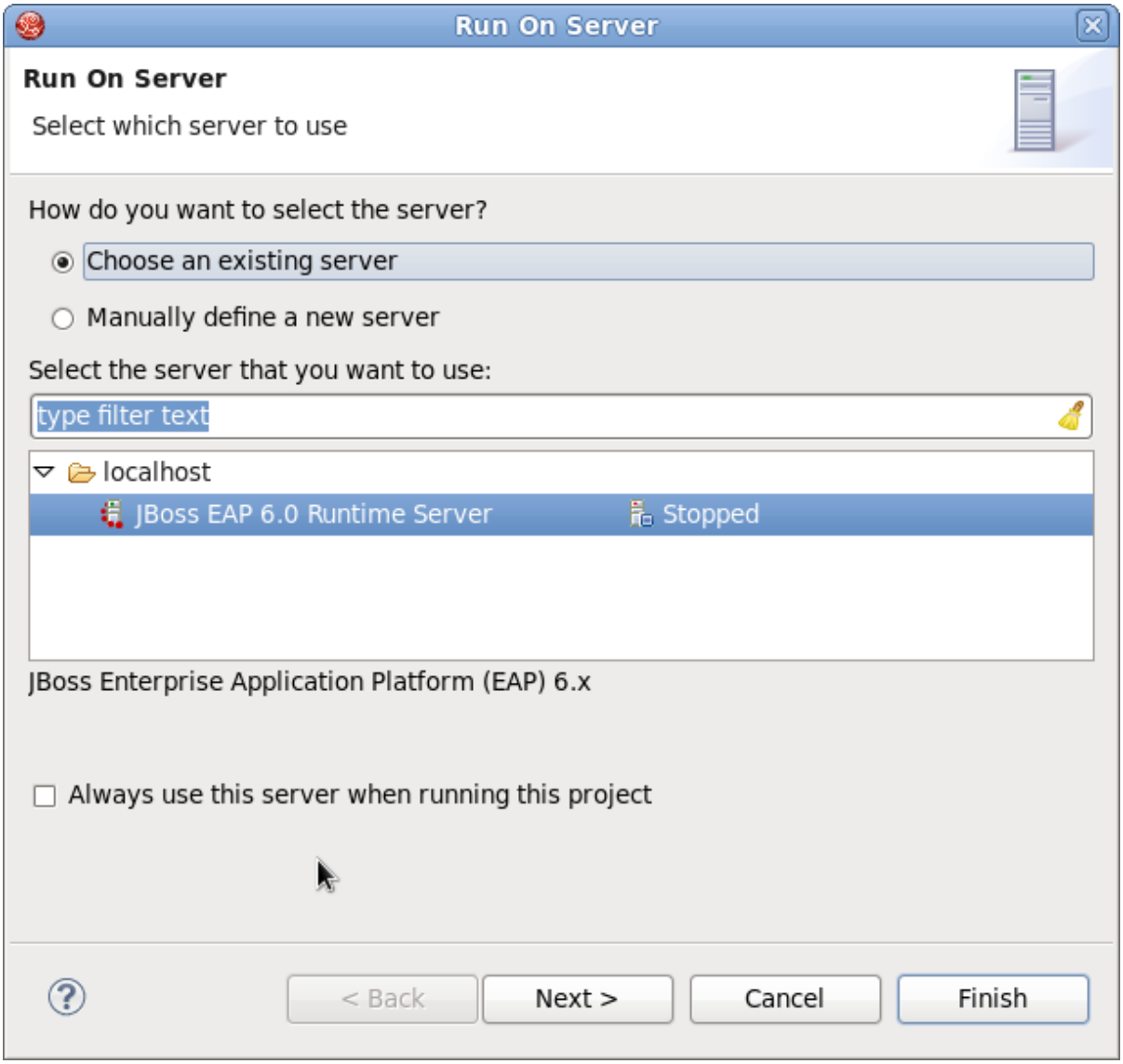


図1.11 サーバー上での実行

3. **[JBoss EAP 6.0 Runtime Server]** サーバーを選択し、**[Next]** をクリックします。これにより **helloworld** クイックスタートが JBoss サーバーにデプロイされます。

4. **helloworld** が JBoss サーバーに正しくデプロイされたことを確認するには、Web ブラウザーを開き、URL http://localhost:8080/jboss-as-helloworld を指定してアプリケーションにアクセスします。

手順1.11 ディレクトリ構造の確認

helloworld クイックスタートのコードは ***QUICKSTART_HOME*/helloworld** ディレクトリにあります。**helloworld** クイックスタートはサーブレットと CDI Bean によって構成されます。また、このアプリケーションの Bean を検索し、CDI をアクティベートするよう JBoss Enterprise Application Platform 6 に伝える空の beans.xml ファイルも含まれています。

1. **beans.xml** はクイックスタートの **src/main/webapp/** ディレクトリにある **WEB-INF/** フォルダーにあります。

2. **src/main/webapp/** ディレクトリには、単純なメタリフレッシュを使用してユーザーのブラウザーを http://localhost:8080/jboss-as-helloworld/HelloWorld にあるサーブレットへリダイレクトする **index.html** ファイルも含まれています。

3. この例の全設定は、例の **src/main/webapp/** ディレクトリにある **WEB-INF/** にあります。

4. クイックスタートには **web.xml** ファイルは必要ありません。

手順1.12 コードの確認

パッケージの宣言とインポートはこれらのリストには含まれていません。完全なリストはクイックスタートのソースコードにあります。

1. **HelloWorldServlet コードの検証**
   HelloWorldServlet.java は **src/main/java/org/jboss/as/quickstarts/helloworld/** ディレクトリにあります。このサーブレットが情報をブラウザーに送ります。

```
27. @WebServlet("/HelloWorld")
28. public class HelloWorldServlet extends HttpServlet {
29.
30.    static String PAGE_HEADER = "<html><head /><body>";
31.
32.    static String PAGE_FOOTER = "</body></html>";
33.
34.    @Inject
35.    HelloService helloService;
36.
37.    @Override
38.    protected void doGet(HttpServletRequest req,
HttpServletResponse resp)
                              throws ServletException, IOException
{
39.       PrintWriter writer = resp.getWriter();
40.       writer.println(PAGE_HEADER);
41.       writer.println("<h1>" +
helloService.createHelloMessage("World") + "</h1>");
42.       writer.println(PAGE_FOOTER);
43.       writer.close();
44.     }
45.
46. }
```

表1.1 HelloWorldServlet の詳細

行	注記
27	Java EE 6 以前はサーブレットの登録に XML ファイルが使用されました。サーブレットの登録はかなり簡易化され、**@WebServlet** アノテーションを追加し、サーブレットへのアクセスに使用される URL へのマッピングを提供することのみが必要となります。
30-32	各 Web ページには適切な形式の HTML が必要になります。本クイックスタートは静的な文字列を使用して最低限のヘッダーとフッターの出力を書き出します。

行	注記
34-35	これらの行は実際のメッセージを生成する HelloService CDI Bean を挿入します。HelloService の API を変更しない限り、ビューレイヤーを変更せずに HelloService の実装を後日変更することが可能です。
41	この行はサービスへ呼び出し、「Hello World」というメッセージを生成して HTTP 要求へ書き出します。

2. **HelloService コードの検証**
   `HelloService.java` ファイルは **`src/main/java/org/jboss/as/quickstarts/helloworld/`** ディレクトリにあります。このサービスは非常に単純であり、メッセージを返します。XML やアノテーションの登録は必要ありません。

```
 9. public class HelloService {
10.
11.    String createHelloMessage(String name) {
12.       return "Hello " + name + "!";
32.    }
33. }
34.
```

バグを報告する

1.4.4.2. numberguess クイックスタート

概要

このクイックスタートでは単純なアプリケーションを作成し、 JBoss Enterprise Application Platform 6 にデプロイする方法を説明します。ここで作成するアプリケーションは情報を永続化しません。情報は JSF ビューを使用して表示され、ビジネスロジックは 2 つの CDI (Contexts and Dependency Injection) Bean にカプセル化されます。**numberguess** クイックスタートでは 1 から 100 までの数字を当てるチャンスが 10 回与えられます。数字を選択した後、その数字が正解の数字より大きいか小さいかが表示されます。

numberguess クイックスタートのコードは ***`QUICKSTART_HOME`*`/numberguess`** ディレクトリにあります。**numberguess** クイックスタートは WAR モジュールとしてパッケージ化された複数の Bean や設定ファイル、Facelets (JSF) ビューによって構成されます。

コマンドラインを使用してこのクイックスタートをビルドしデプロイする手順の詳細は **numberguess** クイックスタートディレクトリのルートにある README ファイルを参照してください。ここでは JBoss Developer Studio を使用してクイックスタートを実行する方法を説明します。

手順1.13 numberguess クイックスタートを JBoss Developer Studio にインポートします。

「JBoss Developer Studio でのクイックスタートの実行」に記述された手順に従ってすでにすべてのクイックスタートを JBoss Developer Studio にインポートした場合は、次のセクションに進むことができます。

1. JBoss Developer Studio がインストールされていない場合は、 「JBoss Developer Studio 5 のインストール」に記述された手順に従います。

2. 「JBoss Developer Studio の起動」に記述された手順に従います。

3. メニューより **[File]** → **[Import]** と選択します。

4. 選択リストより **[Maven]** → **[Maven Projects]** と選択し、**[Next]** を選択します。

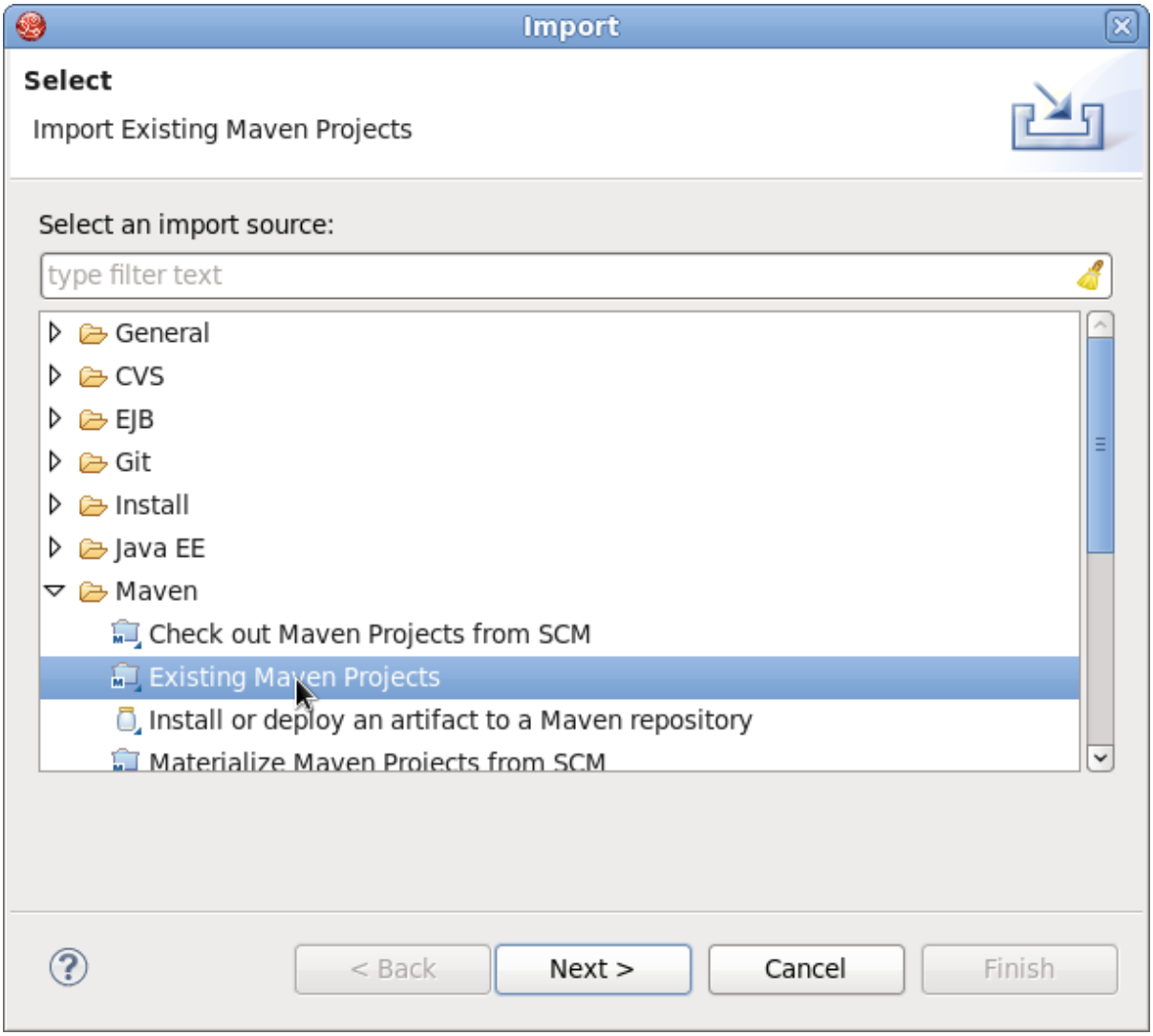


図1.12 既存の Maven プロジェクトのインポート

5. ***QUICKSTART_HOME*/quickstart/numberguess/** ディレクトリを閲覧し、**[OK]** をクリックします。**[Projects]** リストボックスに **[numberguess]** クイックスタートプロジェクトから **pom.xml** ファイルが追加されます。

6. **[Finish]** をクリックします。

手順1.14 numberguess クイックスタートのビルドとデプロイ

1. JBoss Enterprise Application Platform 6 用 JBoss Developer Studio がまだ設定されていない場合は、「JBoss Enterprise Application Platform 6 サーバーの JBoss Developer Studio への追加」に記述された手順に従います。

2. **[Project Explorer]** タブの **[jboss-as-numberguess]** を右クリックし、**[Run As]** → **[Run on Server]** と選択します。

3. **[JBoss EAP 6.0 Runtime Server]** サーバーを選択し、**[Next]** をクリックします。これにより **numberguess** クイックスタートが JBoss サーバーにデプロイされます。

4. **numberguess** が JBoss サーバーに正しくデプロイされたことを確認するには、Web ブラウザを開き、URL http://localhost:8080/jboss-as-numberguess を指定してアプリケーションにアクセスします。

手順1.15 設定ファイルの確認

この例の設定ファイルはすべてクイックスタートの **src/main/webapp/** ディレクトリにある **WEB-INF/** ディレクトリに格納されています。

1. **faces-config ファイルの確認**
   本クイックスタートは **faces-config.xml** ファイル名の JSF 2.0 バージョンを使用します。Facelets の標準的なバージョンが JSF 2.0 のデフォルトのビューハンドラーであるため、特に必要なものはありません。ここでは JBoss Enterprise Application Platform 6 は Java EE の領域を越えます。この設定ファイルが含まれると JSF が自動的に設定されます。そのため、設定はルート要素のみで構成されます。

```
03. <faces-config version="2.0"
04.     xmlns="http://java.sun.com/xml/ns/javaee"
05.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
06.     xsi:schemaLocation="
07.        http://java.sun.com/xml/ns/javaee>
08.        http://java.sun.com/xml/ns/javaee/web-
facesconfig_2_0.xsd">
09.
10. </faces-config>
```

2. **beans.xml ファイルの確認**
   アプリケーションの Bean を検索し、CDI を有効にするよう JBoss Enterprise Application Platform に伝える空の **beans.xml** ファイルも存在します。

3. **web.xml ファイルはありません**
   クイックスタートには **web.xml** ファイルは必要ありません。

手順1.16 JSF コードの確認

JSF はソースファイルに **.xhtml** ファイル拡張子を使用しますが、レンダリングされたビューには **.jsf** 拡張子を使用します。

- **home.xhtml コードの確認**
  home.xhtml ファイルは **src/main/webapp/** ディレクトリにあります。

```
03. <html xmlns="http://www.w3.org/1999/xhtml"
04.     xmlns:ui="http://java.sun.com/jsf/facelets"
05.     xmlns:h="http://java.sun.com/jsf/html"
06.     xmlns:f="http://java.sun.com/jsf/core">
07.
08. <head>
09. <meta http-equiv="Content-Type" content="text/html; charset=iso-
8859-1" />
10. <title>Numberguess</title>
11. </head>
12.
13. <body>
```

```
14.    <div id="content">
15.       <h1>Guess a number...</h1>
16.       <h:form id="numberGuess">
17.
18.          <!-- Feedback for the user on their guess -->
19.          <div style="color: red">
20.             <h:messages id="messages" globalOnly="false" />
21.             <h:outputText id="Higher" value="Higher!"
22.                rendered="#{game.number gt game.guess and
game.guess ne 0}" />
23.             <h:outputText id="Lower" value="Lower!"
24.                rendered="#{game.number lt game.guess and
game.guess ne 0}" />
25.          </div>
26.
27.          <!-- Instructions for the user -->
28.          <div>
29.             I'm thinking of a number between <span
30.                id="numberGuess:smallest">#
{game.smallest}</span> and <span
31.                id="numberGuess:biggest">#{game.biggest}</span>.
You have
32.             #{game.remainingGuesses} guesses remaining.
33.          </div>
34.
35.          <!-- Input box for the users guess, plus a button to
submit, and reset -->
36.          <!-- These are bound using EL to our CDI beans -->
37.          <div>
38.             Your guess:
39.             <h:inputText id="inputGuess" value="#{game.guess}"
40.                required="true" size="3"
41.                disabled="#{game.number eq game.guess}"
42.                validator="#{game.validateNumberRange}" />
43.             <h:commandButton id="guessButton" value="Guess"
44.                action="#{game.check}"
45.                disabled="#{game.number eq game.guess}" />
46.          </div>
47.          <div>
48.             <h:commandButton id="restartButton" value="Reset"
49.                action="#{game.reset}" immediate="true" />
50.          </div>
51.       </h:form>
52.
53.    </div>
54.
55.    <br style="clear: both" />
56.
57. </body>
58. </html>
```

表1.2 JSF の詳細

行	注記
20-24	ユーザーに送信できるメッセージ、「Higher」(より大きい) と「Lower」(より小さい) になります。
29-32	ユーザーが数を選択するごとに数字の範囲が狭まります。有効な数の範囲が分かるようにこの文章は変更されます。
38-42	このフィールドは値式を使用して Bean プロパティーにバインドされます。
42	ユーザーが誤って範囲外の数字を入力しないようバリデーターのバインディングが使用されます。バリデーターがない場合は、ユーザーが範囲外の数字を使用することがあります。
43-45	ユーザーの選択した数字をサーバーに送る方法があるはずです。ここでは Bean 上のアクションメソッドをバインドします。

手順1.17 クラスファイルの確認

numberguess クイックスタートのソースファイルはすべて **src/main/java/org/jboss/as/quickstarts/numberguess/** ディレクトリにあります。パッケージの宣言とインポートはリストには含まれていません。完全なリストはクイックスタートのソースコードにあります。

1. **Random.java 限定子コードの検証**
   型に基づき挿入の対象となる 2 つの Bean 間の曖昧さを取り除くために修飾子が使用されます。修飾子の詳細については、 「修飾子を使用して不明な挿入を解決」を参照してください。

   @Random 限定子は乱数の挿入に使用されます。

```
21. @Target({ TYPE, METHOD, PARAMETER, FIELD })
22. @Retention(RUNTIME)
23. @Documented
24. @Qualifier
25. public @interface Random {
26.
27. }
```

2. **MaxNumber.java 限定子コードの検証**
   @MaxNumberqualifier は最大許可数の挿入に使用されます。

```
21. @Target({ TYPE, METHOD, PARAMETER, FIELD })
22. @Retention(RUNTIME)
23. @Documented
24. @Qualifier
25. public @interface MaxNumber {
26.
27. }
```

3. **ジェネレーターコードの検証**
   Generator クラスは **producer** メソッドより乱数を作成する役割があります。また、**producer** メソッドより最大可能数も公開します。このクラスはアプリケーションスコープ指定であるため、毎回異なる乱数になることはありません。

```
28. @ApplicationScoped
29. public class Generator implements Serializable {
30.    private static final long serialVersionUID =
-7213673465118041882L;
31.
32.    private java.util.Random random = new
java.util.Random(System.currentTimeMillis());
33.
34.    private int maxNumber = 100;
35.
36.    java.util.Random getRandom() {
37.       return random;
38.    }
39.
40.    @Produces
41.    @Random
42.    int next() {
43.       // a number between 1 and 100
44.       return getRandom().nextInt(maxNumber - 1) + 1;
45.    }
46.
47.    @Produces
48.    @MaxNumber
49.    int getMaxNumber() {
50.       return maxNumber;
51.    }
52. }
```

4. **ゲームコードの検証**
   セッションスコープ指定クラス **Game** はアプリケーションのプライマリエントリーポイントです。ゲームの設定や再設定、ユーザーが選択する数字のキャプチャーや検証、**FacesMessage** によるユーザーへのフィードバック提供を行う役割があります。コンストラクト後の **lifecycle** メソッドを使用し、**@Random Instance<Integer>** Bean より乱数を読み出してゲームを初期化します。

   このクラスの **@Named** アノテーションを見てください。このアノテーションは式言語 **(EL)** より **Bean** を **JSF** ビューにアクセスできるようにしたい場合のみ必要です。この場合 **#{game}** が **EL** になります。

```
035. @Named
036. @SessionScoped
037. public class Game implements Serializable {
038.
039.    private static final long serialVersionUID =
991300443278089016L;
040.
041.    /**
042.     * The number that the user needs to guess
043.     */
```

```
044.    private int number;
045.
046.    /**
047.     * The users latest guess
048.     */
049.    private int guess;
050.
051.    /**
052.     * The smallest number guessed so far (so we can track the
valid guess range).
053.     */
054.    private int smallest;
055.
056.    /**
057.     * The largest number guessed so far
058.     */
059.    private int biggest;
060.
061.    /**
062.     * The number of guesses remaining
063.     */
064.    private int remainingGuesses;
065.
066.    /**
067.     * The maximum number we should ask them to guess
068.     */
069.    @Inject
070.    @MaxNumber
071.    private int maxNumber;
072.
073.    /**
074.     * The random number to guess
075.     */
076.    @Inject
077.    @Random
078.    Instance<Integer> randomNumber;
079.
080.    public Game() {
081.    }
082.
083.    public int getNumber() {
084.       return number;
085.    }
086.
087.    public int getGuess() {
088.       return guess;
089.    }
090.
091.    public void setGuess(int guess) {
092.       this.guess = guess;
093.    }
094.
095.    public int getSmallest() {
096.       return smallest;
097.    }
098.
```

```
099.    public int getBiggest() {
100.       return biggest;
101.    }
102.
103.    public int getRemainingGuesses() {
104.       return remainingGuesses;
105.    }
106.
107.    /**
108.     * Check whether the current guess is correct, and update
the biggest/smallest guesses as needed.
109.     * Give feedback to the user if they are correct.
110.     */
111.    public void check() {
112.       if (guess > number) {
113.          biggest = guess - 1;
114.       } else if (guess < number) {
115.          smallest = guess + 1;
116.       } else if (guess == number) {
117.          FacesContext.getCurrentInstance().addMessage(null, new
FacesMessage("Correct!"));
118.       }
119.       remainingGuesses--;
120.    }
121.
122.    /**
123.     * Reset the game, by putting all values back to their
defaults, and getting a new random number.
124.     * We also call this method when the user starts playing for
the first time using
125.     * {@linkplain PostConstruct @PostConstruct} to set the
initial values.
126.     */
127.    @PostConstruct
128.    public void reset() {
129.       this.smallest = 0;
130.       this.guess = 0;
131.       this.remainingGuesses = 10;
132.       this.biggest = maxNumber;
133.       this.number = randomNumber.get();
134.    }
135.
136.    /**
137.     * A JSF validation method which checks whether the guess is
valid. It might not be valid because
138.     * there are no guesses left, or because the guess is not in
range.
139.     *
140.     */
141.    public void validateNumberRange(FacesContext context,
UIComponent toValidate, Object value) {
142.       if (remainingGuesses <= 0) {
143.          FacesMessage message = new FacesMessage("No guesses
left!");
144.          context.addMessage(toValidate.getClientId(context),
message);
```

```
145.          ((UIInput) toValidate).setValid(false);
146.          return;
147.       }
148.       int input = (Integer) value;
149.
150.       if (input < smallest || input > biggest) {
151.          ((UIInput) toValidate).setValid(false);
152.
153.          FacesMessage message = new FacesMessage("Invalid
guess");
154.          context.addMessage(toValidate.getClientId(context),
message);
155.       }
156.    }
157. }
```

バグを報告する

第2章 MAVEN ガイド

2.1. MAVEN について

2.1.1. Maven リポジトリについて

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

Maven は、リポジトリを使いアーカイブを行います。Maven リポジトリには Java ライブラリ、プラグイン、その他のアーティファクトが格納されています。デフォルトのパブリックリポジトリは Maven 2 Central Repository ですが、複数の開発チームの間で共通のアーティファクトを共有する目的で、社内のプライベートおよび内部リポジトリとすることが可能です。また、サードパーティのリポジトリもあります。JBoss Enterprise Application Platform 6 には、Java EE 開発者が通常 JBoss Enterprise Application Platform 6 でアプリケーションを構築する際に利用する要件の多くが含まれています。このようなリポジトリを使うようプロジェクトを設定するには、 「 JBoss Enterprise Application Platform の Maven リポジトリの設定」 を参照してください。

リポジトリはリモートにもローカルにもすることができます。リモートのリポジトリには、HTTP サーバーのリポジトリには**`http://`**、ファイルサーバーのリポジトリには**`file://`** という風に共通のプロトコルを使いアクセスします。ローカルのリポジトリは、リモートリポジトリからのアーティファクトをダウンロードしキャッシュ化したものです。

Maven に関する詳細情報は、Welcome to Apache Maven を参照してください。

また、Maven リポジトリの情報は Apache Maven Project - Introduction to Repositories を確認してください。

さらに、Maven POM ファイルの詳細情報は Apache Maven Project POM Reference および 「Maven POM ファイルについて」 から確認いただけます。

バグを報告する

2.1.2. Maven POM ファイルについて

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

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

POM ファイルの詳細は Apache Maven Project POM Reference を参照してください。

バグを報告する

2.1.3. Maven POM ファイルの最低要件

最低要件

pom.xml ファイルの最低要件は次の通りです。

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

サンプル pom.xml ファイル

基本的な **pom.xml** ファイルは次のようになります。

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

バグを報告する

2.1.4. Maven 設定ファイルについて

Maven の **settings.xml** ファイルには Maven に関するユーザー固有の設定情報が含まれています。開発者の ID、プロキシ情報、ローカルリポジトリの場所など、 **pom.xml** ファイルで配布されてはならないユーザー固有の設定が含まれています。

settings.xml が存在する場所は 2 つあります。

Maven インストール

設定ファイルは ***M2_HOME*/conf/** ディレクトリにあります。これらの設定は**global** 設定と呼ばれます。デフォルトの Maven 設定ファイルはコピー可能なテンプレートで、これを基にユーザー設定ファイルを設定することが可能です。

ユーザーのインストール

設定ファイルは ***USER_HOME*/.m2/** ディレクトリにあります。 Maven とユーザーの **settings.xml** ファイルが存在する場合、内容はマージされます。重複する内容がある場合、ユーザーの **settings.xml** ファイルが優先されます。

Maven **settings.xml** ファイルの例は以下のとおりです。

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <profiles>
```

```
    <!-- Configure the JBoss EAP Maven repository -->
    <profile>
      <id>jboss-eap-maven-repository</id>
      <repositories>
        <repository>
          <id>jboss-eap</id>
          <url>file:///path/to/repo/jboss-eap-6.0-maven-repository</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>jboss-eap-maven-plugin-repository</id>
          <url>file:///path/to/repo/jboss-eap-6.0-maven-repository</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
  <activeProfiles>
    <!-- Optionally, make the repository active by default -->
    <activeProfile>jboss-eap-maven-repository</activeProfile>
  </activeProfiles>
</settings>
```

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

バグを報告する

2.2. MAVEN と JBOSS MAVEN レポジトリのインストール

2.2.1. Maven のダウンロードとインストール

1. Apache Maven Project - Download Maven へアクセスし、ご使用のオペレーティングシステムに対する最新のディストリビューションをダウンロードします。
2. ご使用のオペレーシングシステムに対して Apache Maven をダウンロードしインストールする方法については Maven のドキュメントを参照してください。

バグを報告する

2.2.2. JBoss Enterprise Application Platform 6 の Maven リポジトリのインストール

リポジトリをインストールする方法には、ローカルファイルシステム上のインストール、Apache Web サーバー上のインストール、Maven リポジトリマネージャーを使用したインストールの 3 つの方法があります。

- 「JBoss Enterprise Application Platform 6 の Maven リポジトリのローカルインストール」
- 「Apache httpd を使用するため JBoss Enterprise Application Platform 6 の Maven リポジトリをインストールする」
- 「Nexus Maven リポジトリマネージャーを使用して JBoss Enterprise Application Platform 6 の Maven リポジトリをインストールする」

バグを報告する

2.2.3. JBoss Enterprise Application Platform 6 の Maven リポジトリのローカルインストール

概要

リポジトリをインストールする方法には、ローカルファイルシステム上のインストール、Apache Web サーバー上のインストール、Maven リポジトリマネージャーを使用したインストールの 3 つの方法があります。この例では、ローカルのファイルシステムへ JBoss Enterprise Application Platform 6 の Maven リポジトリをダウンロードする手順を取り上げます。このオプションは設定が簡単で、ローカルマシンですぐ使用することが可能になります。開発環境で Maven の知識を深めることができますが、チームによる実稼働環境での使用は推奨されません。

手順2.1 JBoss Enterprise Application Platform 6 Maven リポジトリーをダウンロードしてローカルファイルシステムにインストールする

1. **JBoss Enterprise Application Platform 6 の Maven リポジトリ ZIP アーカイブのダウンロード**
   Web ブラウザーを開き、URL https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=appplatform にアクセスします。
2. リストに「Application Platform 6 Maven リポジトリ」があることを確認します。
3. **[ダウンロード]** ボタンをクリックし、リポジトリが含まれる **.zip** ファイルをダウンロードします。
4. ローカルファイルシステム上の同じディレクトリにあるファイルを希望のディレクトリへ解凍します。
5. 「Maven 設定を使用した JBoss Enterprise Application Platform の Maven リポジトリーの設定」.

結果

これにより、**jboss-eap-6.1.0.maven-repository** という名前の Maven リポジトリーディレクトリーが作成されます。

重要

古いローカルリポジトリーを引き続き使用する場合は、そのリポジトリーを Maven **settings.xml** 設定ファイルで個別に設定する必要があります。各ローカルリポジトリーは、独自の **<repository>** タグ内で設定する必要があります。

重要

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

バグを報告する

2.2.4. Apache httpd を使用するため JBoss Enterprise Application Platform 6 の Maven リポジトリをインストールする

リポジトリをインストールする方法には、ローカルファイルシステム上のインストール、Apache Web サーバー上のインストール、Maven リポジトリマネージャーを使用したインストールの 3 つの方法があります。この例では、Apache httpd を使用するため JBoss Enterprise Application Platform 6 の Maven リポジトリをダウンロードする手順を取り上げます。Web サーバーにアクセスできる開発者は Maven リポジトリにもアクセスできるため、このオプションは マルチユーザーの開発環境や、チームにまたがる開発環境向けのオプションになります。

要件

Apache httpd を設定する必要があります。手順は Apache HTTP Server Project を参照してください。

手順2.2 JBoss Enterprise Application Platform 6 の Maven リポジトリ ZIP アーカイブのダウンロード

1. Web ブラウザーを開き、URL https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=appplatform にアクセスします。
2. リストに「Application Platform 6 Maven リポジトリ」があることを確認します。
3. **[ダウンロード]** ボタンをクリックし、リポジトリが含まれる **.zip** ファイルをダウンロードします。
4. Apache サーバー上で Web にアクセス可能なディレクトリにファイルを展開します。
5. Apache を設定し、作成されたディレクトリの読み取りアクセスとディレクトリの閲覧を許可します。
6. 「Maven 設定を使用した JBoss Enterprise Application Platform の Maven リポジトリーの設定」に記載された手順に従います。

結果

マルチユーザー環境が Apache httpd 上で Maven リポジトリにアクセスできるようになります。

注記

以前のバージョンのリポジトリーからアップグレードする場合は、競合を発生させずに JBoss Enterprise Application Platform 6.1.0 Maven リポジトリーアーティファクトを既存の JBoss 製品 Maven リポジトリー (JBoss Enterprise Application Platform 6.0.1 など) に単純に抽出できることに注意してください。リポジトリーアーカイブの抽出後は、このリポジトリーの既存の Maven 設定でアーティファクトを使用できます。

バグを報告する

2.2.5. Nexus Maven リポジトリマネージャーを使用して JBoss Enterprise Application Platform 6 の Maven リポジトリをインストールする

リポジトリをインストールする方法には、ローカルファイルシステム上のインストール、Apache Web サーバー上のインストール、Maven リポジトリマネージャーを使用したインストールの 3 つの方法があります。Nexus Maven リポジトリマネージャーを使用すると、JBoss リポジトリを既存のリポジトリと共にホストできるため、ライセンスを所有し、既にリポジトリマネージャーを使用している場合はこの方法が最適です。Maven リポジトリマネージャーの詳細は 「Maven リポジトリマネージャーについて」 を参照してください。

この例では Sonatype Nexus Maven リポジトリマネージャーを使用して JBoss Enterprise Application Platform 6 の Maven リポジトリをインストールする手順を取り上げます。 完全な手順は Sonatype Nexus: Manage Artifacts を参照してください。

手順2.3 JBoss Enterprise Application Platform 6 の Maven リポジトリ ZIP アーカイブのダウンロード

1. Web ブラウザーを開き、URL https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=appplatform にアクセスします。
2. リストに「Application Platform 6 Maven リポジトリ」があることを確認します。
3. **[ダウンロード]** ボタンをクリックし、リポジトリが含まれる **.zip** ファイルをダウンロードします。
4. 希望のディレクトリにファイルを展開します。

手順2.4 Nexus Maven リポジトリマネージャーを使用して JBoss Enterprise Application Platform 6 の Maven リポジトリを追加する

1. 管理者として Nexus にログインします。
2. リポジトリマネージャーの左側の **[Views]** → **[Repositories]** メニューより **[Repositories]** セクションを選択します。
3. **[Add...]** ドロップダウンメニューをクリックし、**[Hosted Repository]** を選択します。
4. 新しいリポジトリに名前と ID をつけます。
5. フィールド **[Override Local Storage]** の場所に、展開されたリポジトリへのディスク上のパスを入力します。
6. リポジトリグループでアーティファクトを使用できるようにする場合は次の手順に従い設定を継続します。必要がない場合は継続しないでください。
7. リポジトリグループを選択します。
8. **[Configure]** タブをクリックします。
9. **[Available Repositories]** リストにある新しい JBoss Maven リポジトリを左側の **[Ordered Group Repositories]** へドラッグします。

注記

このリストの順番により Maven アーティファクトの検索優先度が決定されます。

10. 「Maven 設定を使用した JBoss Enterprise Application Platform の Maven リポジトリーの設定」に記載された手順に従います。

結果

Nexus Maven リポジトリマネージャーを使用してリポジトリが設定されます。

バグを報告する

2.2.6. Maven リポジトリマネージャーについて

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

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

Maven リポジトリーマネージャーの詳細については、「Apache Maven Project - The List of Repository Managers (Apache Maven プロジェクト - リポジトリーマネージャーのリスト)」を参照してください。

一般的に使用される Maven リポジトリマネージャー

Sonatype Nexus

Nexus に関する詳しい情報は Sonatype Nexus: Manage Artifacts を参照してください。

Artifactory

Artifactory に関する詳しい情報は Artifactory Open Source を参照してください。

Apache Archiva

Apache Archiva に関する詳しい情報は Apache Archiva: The Build Artifact Repository Manager を参照してください。

バグを報告する

2.3. MAVEN レポジトリの設定

2.3.1. JBoss Enterprise Application Platform の Maven リポジトリの設定

概要

プロジェクトで Maven により JBoss Enterprise Application Platform Maven リポジトリーを使用するには 2 つの方法があります。

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

手順2.5 JBoss Enterprise Application Platform の Maven リポジトリを使用するよう Maven を設定する

1. **Maven の設定を使用して Maven リポジトリを設定する**
   これは推奨される方法です。リポジトリマネージャーや共有サーバーを用いたリポジトリを使用して Maven を設定すると、プロジェクトの制御や管理が向上します。また、代替のミラーを使用してプロジェクトファイルを変更せずにリポジトリマネージャーへの特定リポジトリのルックアップ要求をすべてリダイレクトすることが可能になります。ミラーに関する詳細については、http://maven.apache.org/guides/mini/guide-mirror-settings.html を参照してください。

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

   「Maven 設定を使用した JBoss Enterprise Application Platform の Maven リポジトリーの設定」

2. **プロジェクトの POM を使用して Maven リポジトリを設定する**
   通常、この方法は推奨されません。プロジェクトの POM ファイルにリポジトリーを設定する場合は慎重に計画します。ビルドに時間がかかり、想定外のリポジトリーからアーティファクトが抽出されることがあることに注意してください。

   注記

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

   このような設定では、推移的に含まれた POM も問題になります。Maven は、これらの外部リポジトリーで不明なアーティファクトを問い合わせる必要があります。これにより、ビルドに時間がかかるだけでなく、アーティファクトの抽出元を制御できなくなり、多くの場合、ビルドが破壊されます。

   この設定方法は、設定されたプロジェクトのグローバルおよびユーザーの Maven 設定を上書きします。

   「プロジェクト POM を用いた JBoss Enterprise Application Platform の Maven リポジトリの設定」を参照してください。

バグを報告する

2.3.2. Maven 設定を使用した JBoss Enterprise Application Platform の Maven リポジトリーの設定

プロジェクトにおいて Maven で JBoss Enterprise Application Platform Maven リポジトリーを使用するには 2 つの方法があります。

- Maven 設定を変更できます。
- プロジェクトの POM ファイルを設定できます。

このタスクでは、Maven で Maven グローバル設定またはユーザー設定を使用してすべてのプロジェクトで JBoss Enterprise Application Platform Maven リポジトリーを使用する方法を示します。これは推奨される方法です。

注記

リポジトリーの URL は、リポジトリーが存在する場所 (ファイルシステムまたはWeb サーバー) によって異なります。 リポジトリーのインストール方法については、JBoss Enterprise Application Platform 6 用 『開発ガイド (Development Guide)』 (https://access.redhat.com/site/documentation/JBoss_Enterprise_Application_Platform/ の 『Maven ガイド (Maven Guide)』 を参照してください。各インストールオプションの例は以下のとおりです。

ファイルシステム

```
file:///path/to/repo/jboss-eap-6.0.0-maven-repository
```

Apache Web サーバー

```
http://intranet.acme.com/jboss-eap-6.0.0-maven-repository/
```

Nexus リポジトリマネージャー

```
https://intranet.acme.com/nexus/content/repositories/jboss-eap-6.0.0-maven-repository
```

インストール設定またはユーザーインストール設定を使用して Maven で JBoss Enterprise Application Platform リポジトリーを使用するよう設定できます。設定の場所や動作の詳細については、JBoss Enterprise Application Platform 6 用 『開発ガイド (Development Guide)』 (https://access.redhat.com/site/documentation/JBoss_Enterprise_Application_Platform/) の章 「『Maven ガイド (Maven Guide)』」 を参照してください。

ローカルのユーザーシステムで JBoss Enterprise Application Platform のリポジトリを使用する場合は、次の手順に従ってください。

手順2.6 設定

1. 選択した設定タイプの **settings.xml** を開きます。
   - **グローバル設定**
     global 設定を設定する場合は ***M2_HOME*/conf/settings.xml** ファイルを開きます。
   - **ユーザー設定**
     ユーザー固有の設定を設定する場合、***USER_HOME*/.m2/settings.xml** ファイルが存在しない時は ***M2_HOME*/conf/** ディレクトリの **settings.xml** ファイルを ***USER_HOME*/.m2/** ディレクトリにコピーします。
2. 以下の XML を **settings.xml** ファイルの **<profiles>** 要素にコピーします。**<url>** を実際のリポジトリーの場所に変更します。

```
<profile>
  <id>jboss-eap-repository</id>
  <repositories>
    <repository>
      <id>jboss-eap-repository</id>
      <name>JBoss EAP Maven Repository</name>
      <url>file:///path/to/repo/jboss-eap-6.0.0-maven-
repository</url>
      <layout>default</layout>
      <releases>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
        <enabled>false</enabled>
        <updatePolicy>never</updatePolicy>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>jboss-eap-repository-group</id>
      <name>JBoss EAP Maven Repository</name>
      <url>
      file:///path/to/repo/jboss-eap-6.0.0-maven-repository
      </url>
      <layout>default</layout>
      <releases>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
        <enabled>false</enabled>
        <updatePolicy>never</updatePolicy>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>
```

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

```
<activeProfile>jboss-eap-repository</activeProfile>
```

3. JBoss Developer Studio の稼働中に **settings.xml** ファイルを変更する時は、ユーザー設定をリフレッシュする必要があります。メニューより **[Window]** → **[Preferences]** と選択します。 **[Preferences]** ウインドウで **[Maven]** を開き、**[User Settings]** を選択します。**[Update Settings]** ボタンをクリックし、 JBoss Developer Studio で Maven のユーザー設定をリフレッシュします。

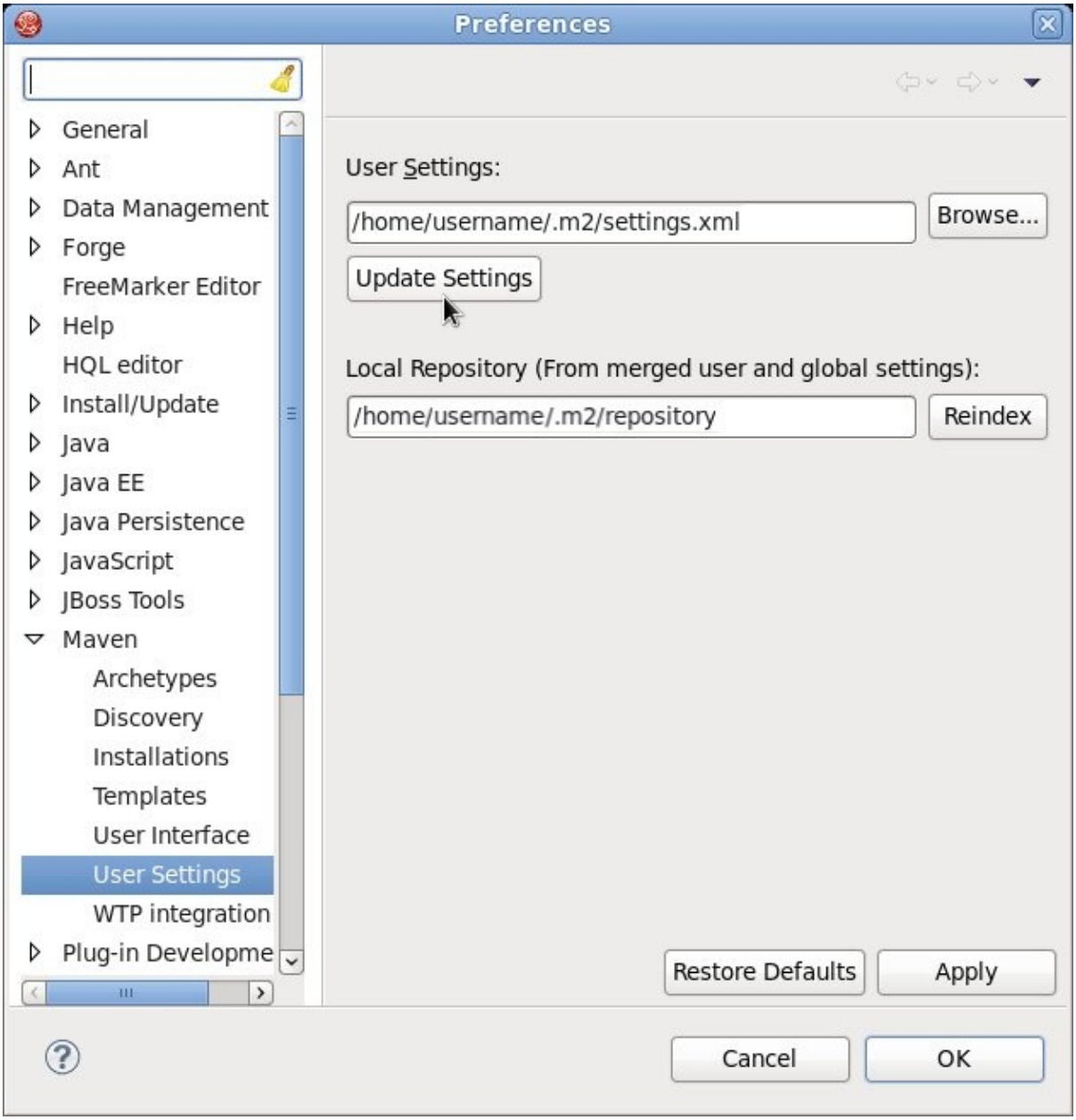


図2.1 Maven ユーザー設定の更新

重要

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

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

この問題を解決するには、最新の Maven アーティファクトをダウンロードするためにローカルリポジトリーのキャッシュバージョンを削除します。キャッシュバージョンは **~/.m2/repository/** サブディレクトリー (Linux の場合) または ***%SystemDrive%*\Users*USERNAME*\.m2\repository** サブディレクトリー (Windows の場合) に存在します。

結果

JBoss Enterprise Application Platform のリポジトリが設定されます。

バグを報告する

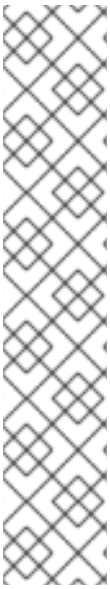
2.3.3. プロジェクト POM を用いた JBoss Enterprise Application Platform の Maven リポジトリの設定

プロジェクトにおいて Maven で JBoss Enterprise Application Platform Maven リポジトリーを使用するには 2 つの方法があります。

- Maven 設定を変更できます。
- プロジェクトの POM ファイルを設定できます。

このタスクでは、リポジトリー情報をプロジェクトの **pom.xml** に追加して、JBoss Enterprise Application Platform の Maven リポジトリーを使用するよう特定のプロジェクトを設定する方法について説明します。この設定方法は、グローバル設定とユーザー設定よりも優先されます。

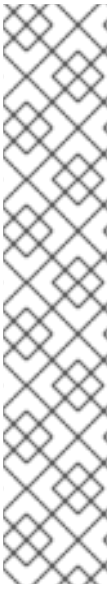
通常、この設定方法は推奨されません。プロジェクトの POM ファイルにリポジトリーを設定する場合は慎重に計画します。ビルドに時間がかかり、想定外のリポジトリーからアーティファクトが抽出されることがあることに注意してください。



注記

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

このような設定では、推移的に含まれた POM も問題になります。Maven は、これらの外部リポジトリーで不明なアーティファクトを問い合わせる必要があります。これにより、ビルドに時間がかかるだけでなく、アーティファクトの抽出元を制御できなくなり、多くの場合、ビルドが破壊されます。



注記

リポジトリーの URL はリポジトリの場所 (ファイルシステムまたは Web サーバー) によって異なります。リポジトリーのインストール方法については、「JBoss Enterprise Application Platform 6 の Maven リポジトリのインストール」を参照してください。各インストールオプションの例は次のとおりです。

ファイルシステム

```
file:///path/to/repo/jboss-eap-6.0.0-maven-repository
```

Apache Web サーバー

```
http://intranet.acme.com/jboss-eap-6.0.0-maven-repository/
```

Nexus リポジトリマネージャー

```
https://intranet.acme.com/nexus/content/repositories/jboss-eap-6.0.0-maven-repository
```

1. テキストエディターでプロジェクトの **pom.xml** ファイルを開きます。

2. 次のリポジトリ設定を追加します。既にファイルに **<repositories>** 設定が存在する場合は **<repository>** 要素を追加します。必ず **<url>** をリポジトリが実存する場所に変更するようにしてください。

```
<repositories>
   <repository>
      <id>jboss-eap-repository-group</id>
      <name>JBoss EAP Maven Repository</name>
      <url>file:///path/to/repo/jboss-eap-6.0.0-maven-
repository/</url>
      <layout>default</layout>
      <releases>
         <enabled>true</enabled>
         <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
         <enabled>true</enabled>
         <updatePolicy>never</updatePolicy>
      </snapshots>
   </repository>
</repositories>
```

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

```
<pluginRepositories>
   <pluginRepository>
      <id>jboss-eap-repository-group</id>
      <name>JBoss EAP Maven Repository</name>
      <url>file:///path/to/repo/jboss-eap-6.0.0-maven-
repository/</url>
      <releases>
         <enabled>true</enabled>
      </releases>
      <snapshots>
         <enabled>true</enabled>
      </snapshots>
   </pluginRepository>
</pluginRepositories>
```

バグを報告する

第3章 クラスローディングとモジュール

3.1. はじめに

3.1.1. クラスロードとモジュールの概要

JBoss Enterprise Application Platform 6 は、デプロイされたアプリケーションのクラスパスを制御するために新しいモジュール形式のクラスロードシステムを使用します。このシステムでは、階層クラスローダーの従来のシステムよりも、柔軟性と制御が強化されています。開発者は、アプリケーションで利用可能なクラスに対して粒度の細かい制御を行い、アプリケーションサーバーで提供されるクラスを無視して独自のクラスを使用してデプロイメントを設定できます。

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

以下に、開発者が JBoss Enterprise Application Platform 6 でアプリケーションを正しくビルドおよびデプロイするために知る必要があることを示します。

バグを報告する

3.1.2. クラスローディング

クラスローディングとは、Java クラスやリソースを Java ランタイム環境にロードするメカニズムのことです。

バグを報告する

3.1.3. モジュール

モジュールは、クラスのロードと依存関係の管理に使用される、クラスの論理的なグループです。JBoss Enterprise Application Platform 6 では、静的モジュールと動的モジュールと呼ばれる 2 種類のモジュールが存在します。ただし、この 2 種類のモジュールの違いは、パッケージ化の方法のみです。すべてのモジュールは同じ機能を提供します。

静的モジュール

静的モジュールは、アプリケーションサーバーの ***EAP_HOME*/modules/** ディレクトリに事前定義されます。各サブディレクトリは 1 つのモジュールを表し、1 つまたは複数の JAR ファイルと設定ファイル (**module.xml**) が含まれます。モジュールの名前は、**module.xml** ファイルで定義されます。アプリケーションサーバーで提供されるすべての API (Java EE API や JBoss Logging などの他の API を含む) は、静的モジュールとして提供されます。

カスタム静的モジュールの作成は、同じサードパーティーライブラリを使用する同じサーバー上に多くのアプリケーションがデプロイされる場合に役立ちます。これらのライブラリを各アプリケーションとバンドルする代わりに、JBoss 管理者はこれらのライブラリが含まれるモジュールを作成およびインストールできます。アプリケーションは、カスタム静的モジュールで明示的な依存関係を宣言できます。

動的モジュール

動的モジュールは、各 JAR または WAR デプロイメント (または、EAR 内のサブデプロイメント) に対してアプリケーションサーバーによって作成およびロードされます。動的モジュールの名前は、

デプロイされたアーカイブの名前から派生されます。デプロイメントはモジュールとしてロードされるため、依存関係を設定でき、他のデプロイメントが依存関係として使用できます。

モジュールは必要なときのみロードされます。通常、明示的または暗黙的な依存関係があるアプリケーションがデプロイされるときのみ、モジュールがロードされます。

バグを報告する

3.1.4. モジュールの依存関係

モジュール依存関係とは、あるモジュールが機能するには別のモジュールのクラスを必要とする宣言のことです。モジュールはいくつでも他のモジュールの依存関係を宣言することができます。アプリケーションサーバーがモジュールをロードする時、モジュールクラスローダーがモジュールの依存関係を解析し、各依存関係のクラスをクラスパスに追加します。指定の依存関係が見つからない場合、モジュールはロードできません。

デプロイされたアプリケーション (JAR および WAR) は動的モジュールとしてロードされ、依存関係を用いて JBoss Enterprise Application Platform 6 によって提供される API へアクセスします。

依存関係には明示的と暗黙的の 2 つのタイプがあります。

明示的な依存関係は開発者によって設定に宣言されます。静的モジュールは依存関係を modules.xml ファイルに宣言することができます。動的モジュールはデプロイメントの MANIFEST.MF または jboss-deployment-structure.xml 記述子に依存関係を宣言することができます。

暗示的な依存関係は、任意の依存関係として指定することができます。任意の依存関係をロードできなくても、モジュールのロードに失敗する原因にはなりません。しかし、依存関係が後で使用できるようになっても、モジュールのクラスパスには追加されません。モジュールがロードされる時に依存関係が使用できなければなりません。

暗黙的な依存関係は、特定の条件やメタデータがデプロイメントで見つかった場合に自動的に追加されます。JBoss Enterprise Application Platform に含まれる Java EE 6 API は、デプロイメントで暗黙的な依存関係が検出された時に追加されるモジュールの一例になります。

デプロイメントを設定して特定の暗黙的な依存関係を除外することも可能です。この設定は jboss-deployment-structure.xml デプロイメント記述子ファイルで行います。これは、アプリケーションサーバーが暗黙的な依存関係として追加しようとする特定バージョンのライブラリを、アプリケーションがバンドルする場合に一般的に行われます。

モジュールのクラスパスには独自のクラスとその直接の依存関係のみが含まれます。モジュールは 1 つの依存関係の依存関係クラスにはアクセスできませんが、暗示的な依存関係のエクスポートを指定できます。エクスポートされた依存関係は、エクスポートするモジュールに依存するモジュールへ提供されます。

> **例3.1 モジュールの依存関係**
>
> モジュール A はモジュール B に依存し、モジュール B はモジュール C に依存します。モジュール A はモジュール B のクラスにアクセスでき、モジュール B はモジュール C のクラスにアクセスできます。以下の場合を除き、モジュール A はモジュール C のクラスにはアクセスできません。
>
> - モジュール A がモジュール C への明示的な依存関係を宣言する場合。
> - または、モジュール B がモジュール B の依存関係をモジュール C でエクスポートする場合。

バグを報告する

3.1.5. デプロイメントでのクラスローディング

JBoss Enterprise Application Platform では、クラスローディングのためにデプロイメントはすべてモジュールとして処理されます。このようなデプロイメントは動的モジュールと呼ばれます。クラスローディングの動作はデプロイメントのタイプによって異なります。

WAR デプロイメント

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

EAR デプロイメント

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

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

サブデプロイメントモジュール (EAR 内の WAR、JAR デプロイメント) は、自動的に親モジュールに依存しますが、サブデプロイメント同士が自動的に依存するわけではありません。これは、サブデプロイメントの分離 (subdeployment isolation) と呼ばれ、デプロイメントごとまたはアプリケーションサーバー全体で無効にすることができます。

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

バグを報告する

3.1.6. クラスローディングの優先順位

JBoss Enterprise Application Platform 6 のモジュラークラスローダーは優先順位の仕組みを利用してクラスローディングの競合が発生しないようにします。

デプロイメント時に、パッケージとクラスの完全リストがデプロイメント毎そして依存性毎に作成されます。この一覧は、クラスローディングの優先順位のルールに従い順番に並べられています。ランタイムにクラスをロードすると、クラスローダーはこの一覧を検索し最初に一致したものをロードします。こうすることで、デプロイメントクラスパスにある同じクラスやパッケージの複数のコピーが競合しないようにします。

クラスローダーは上から順に(降順) クラスをロードします。

1. 暗黙的な依存性

   Java EE API などの、JBoss Enterprise Application Platform 6 が自動的に追加する依存性です。こちらの依存性は、一般的な機能や JBoss Enterprise Application Platform 6 が対応する API が含まれているため、優先順位が最も高くなっています。

   各暗黙的依存性に関する完全な詳細情報は 「暗黙的なモジュール依存関係」 を参照してください。

2. 明示的な依存性

   アプリケーション設定にて手動で追加される依存性です。これらの依存性は、アプリケーションの **MANIFEST.MF** ファイルや、新しくオプションで追加された JBoss の配備記述子 **jboss-deployment-structure.xml** ファイルを使い追加可能です。

   明示的な依存性の追加方法については、「デプロイメントへの明示的なモジュール依存関係の追加」 を参照してください。

3. ローカルリソース

   デプロイメント内にパッケージ化されるクラスファイル (例：WAR ファイルの **WEB-INF/classes** あるいは **WEB-INF/lib** から)

4. デプロイメント間の依存性

   これらは、EAR デプロイメントにある他のデプロイメントの依存関係です。これには、EAR の**lib** ディレクトリーにあるクラス、あるいは他の EJB jar に定義されているクラスが含まれます。

バグを報告する

3.1.7. 動的モジュールの名前付け

すべてのモジュールは JBoss Enterprise Application Platform 6 によってモジュールとしてロードされ、以下の慣例に従って名前が付けられます。

1. WAR および JAR ファイルのデプロイメントは次の形式で名前が付けられます。

   ```
   deployment.DEPLOYMENT_NAME
   ```

   例えば、**inventory.war** のモジュール名は **deployment.inventory.war** となり、**store.jar** のモジュール名は **deployment.store.jar** となります。

2. エンタープライズアーカイブ内のサブデプロイメントは次の形式で名前が付けられます。

   ```
   deployment.EAR_NAME.SUBDEPLOYMENT_NAME
   ```

   たとえば、エンタープライズアーカイブ **accounts.ear** 内にある **reports.war** のサブデプロイメントのモジュール名は **deployment.accounts.ear.reports.war** になります。

バグを報告する

3.1.8. jboss-deployment-structure.xml

jboss-deployment-structure.xml は JBoss Enterprise Application Platform 6 の新しいオプションデプロイメント記述子です。このデプロイメント記述子を使用すると、デプロイメントのクラスローディングを制御できます。

このデプロイメント記述子の XML スキーマは、***EAP_HOME*/docs/schema/jboss-deployment-structure-1_2.xsd** にあります。

バグを報告する

3.2. デプロイメントへの明示的なモジュール依存関係の追加

このタスクでは、アプリケーションへ明示的な依存関係を追加する方法を説明します。明示的なモジュール依存関係をアプリケーションに追加すると、これらのモジュールのクラスをアプリケーションのクラスパスに追加することができます。

一部の依存関係は JBoss Enterprise Application Platform 6 によって自動的にデプロイメントへ追加されます。詳細は 「暗黙的なモジュール依存関係」 を参照してください。

要件

1. モジュール依存関係を追加するソフトウェアプロジェクトが存在する必要があります。
2. 依存関係として追加するモジュールの名前を覚えておく必要があります。JBoss Enterprise Application Platform に含まれる静的モジュールのリストは 「含まれるモジュール」 を参照してください。モジュールが他のデプロイメントである場合は 「動的モジュールの名前付け」 を参照してモジュール名を判断してください。

依存関係を設定する方法は 2 つあります。

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

手順3.1 MANIFEST.MF へ依存関係の設定を追加する

Maven プロジェクトを設定して **MANIFEST.MF** ファイルに必要な依存関係エントリを作成することができます。「Maven を使用した MANIFEST.MF エントリーの生成」 を参照してください。

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

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

3. **任意: 依存関係を任意にする**
   依存関係エントリーのモジュール名に **optional** を付けると、依存関係を任意にすることができます。

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

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

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

手順3.2 jboss-deployment-structure.xml への依存関係設定の追加

1. **jboss-deployment-structure.xml の追加**
   アプリケーションに **jboss-deployment-structure.xml** ファイルが存在しない場合は、**jboss-deployment-structure.xml** という新しいファイルを作成し、プロジェクトに追加します。このファイルは **<jboss-deployment-structure>** がルート要素の XML ファイルです。

   ```
   <jboss-deployment-structure>

   </jboss-deployment-structure>
   ```

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

2. **依存関係セクションの追加**
   <deployment> 要素をドキュメントルート内に作成し、その中に**<dependencies>** 要素を作成します。

3. **モジュール要素の追加**
   依存関係ノード内に各モジュール依存関係に対するモジュール要素を追加します。**name** 属性をモジュールの名前に設定します。

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

4. **任意: 依存関係を任意にする**
   値が **TRUE** のモジュールエントリーに **optional** 属性を追加すると依存関係を任意にすることができます。この属性のデフォルト値は **FALSE** です。

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

5. **任意: 依存関係のエクスポート**
   値が **TRUE** のモジュールエントリーに **export** 属性を追加すると依存関係をエクスポートできます。この属性のデフォルト値は **FALSE** です。

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

例3.2 2 つの依存関係を持つ jboss-deployment-structure.xml

```
<jboss-deployment-structure>

  <deployment>

     <dependencies>
        <module name="org.javassist" />
        <module name="org.apache.velocity" export="TRUE" />
     </dependencies>

  </deployment>

</jboss-deployment-structure>
```

JBoss Enterprise Application Platform 6 はデプロイされた時に、指定されたモジュールからアプリケーションのクラスパスへクラスを追加します。

バグを報告する

3.3. MAVEN を使用した MANIFEST.MF エントリーの生成

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

要件

1. 作業用の Maven プロジェクトが既に存在している必要があります。
2. Maven プロジェクトが JAR、EJB、WAR プラグイン (**maven-jar-plugin**、**maven-ejb-plugin**、 **maven-war-plugin**) のいずれかを使用しなければなりません。
3. プロジェクトのモジュール依存関係の名前を知っていなければなりません。JBoss Enterprise Application Platform 6 に含まれる静的モジュールの一覧は「含まれるモジュール」 を参照してください。モジュールが別のデプロイメントにある場合は 「動的モジュールの名前付け」 を参照してモジュール名を判断してください。

手順3.3 モジュール依存関係が含まれる MANIFEST.MF ファイルの生成

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

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

2. **依存関係の一覧表示**
   モジュール依存関係の一覧を **<Dependencies>** 要素に追加します。**MANIFEST.MF** に依存関係を追加する時と同じ形式を使用します。この形式に関する詳細は 「デプロイメントへの明示的なモジュール依存関係の追加」 を参照してください。

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

3. **プロジェクトの構築**
   Maven アセンブリゴールを用いたプロジェクトの構築

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

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

例3.3 pom.xml の設定されたモジュール依存関係

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

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

バグを報告する

3.4. モジュールが暗黙的にロードされないようにする

このタスクでは、モジュール依存関係のリストを除外するようアプリケーションを設定する方法を説明します。

デプロイ可能なアプリケーションを設定して暗黙的な依存関係がロードされないようにすることが可能です。これは、アプリケーションサーバーより提供される暗黙的な依存関係とは異なるバージョンのライブラリやフレームワークがアプリケーションに含まれる場合に一般的に行われます。

要件

1. モジュール依存関係を除外するソフトウェアプロジェクトが存在する必要があります。
2. 除外するモジュール名を知っている必要があります。暗黙的な依存関係のリストや状態については 「暗黙的なモジュール依存関係」 を参照してください。

手順3.4 jboss-deployment-structure.xml への依存関係除外設定の追加

1. アプリケーションに **jboss-deployment-structure.xml** ファイルが存在しない場合は、**jboss-deployment-structure.xml** という新しいファイルを作成し、プロジェクトに追加しあす。このファイルは **<jboss-deployment-structure>** がルート要素の XML ファイルです。

   ```
   <jboss-deployment-structure>

   </jboss-deployment-structure>
   ```

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

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

```
<deployment>
   <exclusions>

   </exclusions>
</deployment>
```

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

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

例3.4 2 つのモジュールの除外

```
<jboss-deployment-structure>
   <deployment>
      <exclusions>
         <module name="org.javassist" />
         <module name="org.dom4j" />
      </exclusions>
   </deployment>
</jboss-deployment-structure>
```

バグを報告する

3.5. サブシステムをデプロイメントから除外する

概要

ここではサブシステムをデプロイメントより除外するために必要な手順について説明します。**jboss-deployment-structure.xml** 設定ファイルを編集します。サブシステムの除外はサブシステムの削除と同じ影響がありますが、1 つのデプロイメントのみに適用されます。

手順3.5 サブシステムの除外

1. テキストエディターで **jboss-deployment-structure.xml** ファイルを開きます。

2. 次の XML を **<deployment>** タグの中に追加します。

```
<exclude-subsystems>
  <subsystem name="SUBSYSTEM_NAME" />
</exclude-subsystems>
```

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

結果

サブシステムが除外されます。サブシステムのデプロイメントユニットプロセッサがデプロイメント上で実行されないようになります。

例3.5 jboss-deployment-structure.xml ファイルの例

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

バグを報告する

3.6. クラスローディングとサブデプロイメント

3.6.1. エンタープライズアーカイブのモジュールおよびクラスロード

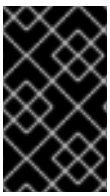
エンタープライズアーカイブ (EAR) は、JAR または WAR デプロイメントのように、単一モジュールとしてロードされません。これらは、複数の一意のモジュールとしてロードされます。

以下のルールによって、EAR に存在するモジュールが決定されます。

- 各 WAR および EJB JAR サブデプロイメントはモジュールです。
- EAR アーカイブのルートにある **lib/** ディレクトリの内容はモジュールです。これは、親モジュールと呼ばれます。

これらのモジュールの動作は、以下の追加の暗黙的な依存関係がある他のモジュールと同じです。

- WAR サブデプロイメントでは、親モジュールとすべての EJB JAR サブデプロイメントに暗黙的な依存関係が存在します。
- EJB JAR サブデプロイメントでは、親モジュールと他のすべての EJB JAR サブデプロイメントに暗黙的な依存関係が存在します。



重要

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

JBoss Enterprise Application Platform 6 ではサブデプロイメントクラスローダーの隔離がデフォルトで無効になるため、上記の暗黙的な依存関係が発生します。

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

バグを報告する

3.6.2. サブデプロイメントクラスローダーの分離

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

サブデプロイメントが他のサブデプロイメントのリソースにアクセスすべきでない場合 (厳格なサブデプロイメントの分離が必要な場合)は、この挙動を有効にできます。

バグを報告する

3.6.3. EAR 内のサブデプロイメントクラスローダーの分離を無効化する

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

重要

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

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

```
<jboss-deployment-structure>

</jboss-deployment-structure>
```

2. **<ear-subdeployments-isolated> 要素の追加**
   <ear-subdeployments-isolated> 要素が存在しない場合は **jboss-deployment-structure.xml** ファイルへ追加し、内容が **false** となるようにします。

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

結果

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

バグを報告する

3.7. 参考資料

3.7.1. 暗黙的なモジュール依存関係

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

表3.1 暗黙的なモジュール依存関係

サブシステム	常に追加されるモジュール	条件付きで追加されるモジュール	条件

サブシステム	常に追加されるモジュール	条件付きで追加されるモジュール	条件
コアサーバー	• **javax.api** • **sun.jdk** • **org.jboss.logging** • **org.apache.log4j** • **org.apache.commons.logging** • **org.slf4j** • **org.jboss.logging.jul-to-slf4j-stub**	-	-
EE サブシステム	• **javaee.api**	-	-
EJB3 サブシステム	-	• **javaee.api**	Java EE 6 の仕様で指定されているように、デプロイメント内の有効な場所で **ejb-jar.xml** が存在するか、アノテーションベースの EJB が存在すること (例：**@Stateless**、**@Stateful**、**@MessageDriven** など)

サブシステム	常に追加されるモジュール	条件付きで追加されるモジュール	条件
JAX-RS (Resteasy) サブシステム	• `javax.xml.bind.api`	• `org.jboss.resteasy.resteasy-atom-provider` • `org.jboss.resteasy.resteasy-cdi` • `org.jboss.resteasy.resteasy-jaxrs` • `org.jboss.resteasy.resteasy-jaxb-provider` • `org.jboss.resteasy.resteasy-jackson-provider` • `org.jboss.resteasy.resteasy-jsapi` • `org.jboss.resteasy.resteasy-multipart-provider` • `org.jboss.resteasy.async-http-servlet-30`	デプロイメント内に JAX-RS のアノテーションが存在すること

サブシステム	常に追加されるモジュール	条件付きで追加されるモジュール	条件
JCA サブシステム	• **javax.resource.api**	• **javax.jms.api** • **javax.validation.api** • **org.jboss.logging** • **org.jboss.ironjacamar.api** • **org.jboss.ironjacamar.impl** • **org.hibernate.validator**	デプロイメントがリソースアダプター (RAR) デプロイメントの場合
JPA (Hibernate) サブシステム	• **javax.persistence.api**	• **javaee.api** • **org.jboss.as.jpa** • **org.hibernate** • **org.javassist**	**@PersistenceUnit** または **@PersistenceContext** アノテーションが存在するか、デプロイメント記述子に **<persistence-unit-ref>** または **<persistence-context-ref>** が存在すること
SAR サブシステム	-	• **org.jboss.logging** • **org.jboss.modules**	デプロイメントが SAR アーカイブであること
セキュリティサブシステム	• **org.picketbox**	-	-

サブシステム	常に追加されるモジュール	条件付きで追加されるモジュール	条件
Web サブシステム	-	• **javaee.api** • **com.sun.jsf-impl** • **org.hibernate.validator** • **org.jboss.as.web** • **org.jboss.logging**	デプロイメントは WAR アーカイブです。利用されている場合は、JavaServer Faces(JSF) のみが追加されます。
Web サービスサブシステム	• **org.jboss.ws.api** • **org.jboss.ws.spi**	-	-
Weld (CDI) サブシステム	-	• **javax.persistence.api** • **javaee.api** • **org.javassist** • **org.jboss.interceptor** • **org.jboss.as.weld** • **org.jboss.logging** • **org.jboss.weld.core** • **org.jboss.weld.api** • **org.jboss.weld.spi**	**beans.xml** ファイルがデプロイメント内で検出された場合

バグを報告する

3.7.2. 含まれるモジュール

- **asm.asm**
- **ch.qos.cal10n**
- **com.google.guava**
- **com.h2database.h2**
- **com.sun.jsf-impl**
- **com.sun.jsf-impl**
- **com.sun.xml.bind**
- **com.sun.xml.messaging.saaj**
- **gnu.getopt**
- **javaee.api**
- **javax.activation.api**
- **javax.annotation.api**
- **javax.api**
- **javax.ejb.api**
- **javax.el.api**
- **javax.enterprise.api**
- **javax.enterprise.deploy.api**
- **javax.faces.api**
- **javax.faces.api**
- **javax.inject.api**
- **javax.interceptor.api**
- **javax.jms.api**
- **javax.jws.api**
- **javax.mail.api**
- **javax.management.j2ee.api**
- **javax.persistence.api**
- **javax.resource.api**

- javax.rmi.api
- javax.security.auth.message.api
- javax.security.jacc.api
- javax.servlet.api
- javax.servlet.jsp.api
- javax.servlet.jstl.api
- javax.transaction.api
- javax.validation.api
- javax.ws.rs.api
- javax.wsdl4j.api
- javax.xml.bind.api
- javax.xml.jaxp-provider
- javax.xml.registry.api
- javax.xml.rpc.api
- javax.xml.soap.api
- javax.xml.stream.api
- javax.xml.ws.api
- jline
- net.sourceforge.cssparser
- net.sourceforge.htmlunit
- net.sourceforge.nekohtml
- nu.xom
- org.antlr
- org.apache.ant
- org.apache.commons.beanutils
- org.apache.commons.cli
- org.apache.commons.codec
- org.apache.commons.collections

- org.apache.commons.io
- org.apache.commons.lang
- org.apache.commons.logging
- org.apache.commons.pool
- org.apache.cxf
- org.apache.httpcomponents
- org.apache.james.mime4j
- org.apache.log4j
- org.apache.neethi
- org.apache.santuario.xmlsec
- org.apache.velocity
- org.apache.ws.scout
- org.apache.ws.security
- org.apache.ws.xmlschema
- org.apache.xalan
- org.apache.xerces
- org.apache.xml-resolver
- org.codehaus.jackson.jackson-core-asl
- org.codehaus.jackson.jackson-jaxrs
- org.codehaus.jackson.jackson-mapper-asl
- org.codehaus.jackson.jackson-xc
- org.codehaus.woodstox
- org.dom4j
- org.hibernate
- org.hibernate.envers
- org.hibernate.infinispan
- org.hibernate.validator
- org.hornetq

- org.hornetq.ra
- org.infinispan
- org.infinispan.cachestore.jdbc
- org.infinispan.cachestore.remote
- org.infinispan.client.hotrod
- org.jacorb
- org.javassist
- org.jaxen
- org.jboss.as.aggregate
- org.jboss.as.appclient
- org.jboss.as.cli
- org.jboss.as.clustering.api
- org.jboss.as.clustering.common
- org.jboss.as.clustering.ejb3.infinispan
- org.jboss.as.clustering.impl
- org.jboss.as.clustering.infinispan
- org.jboss.as.clustering.jgroups
- org.jboss.as.clustering.service
- org.jboss.as.clustering.singleton
- org.jboss.as.clustering.web.infinispan
- org.jboss.as.clustering.web.spi
- org.jboss.as.cmp
- org.jboss.as.connector
- org.jboss.as.console
- org.jboss.as.controller
- org.jboss.as.controller-client
- org.jboss.as.deployment-repository
- org.jboss.as.deployment-scanner

- **org.jboss.as.domain-add-user**
- **org.jboss.as.domain-http-error-context**
- **org.jboss.as.domain-http-interface**
- **org.jboss.as.domain-management**
- **org.jboss.as.ee**
- **org.jboss.as.ee.deployment**
- **org.jboss.as.ejb3**
- **org.jboss.as.embedded**
- **org.jboss.as.host-controller**
- **org.jboss.as.jacorb**
- **org.jboss.as.jaxr**
- **org.jboss.as.jaxrs**
- **org.jboss.as.jdr**
- **org.jboss.as.jmx**
- **org.jboss.as.jpa**
- **org.jboss.as.jpa.hibernate**
- **org.jboss.as.jpa.hibernate**
- **org.jboss.as.jpa.hibernate.infinispan**
- **org.jboss.as.jpa.openjpa**
- **org.jboss.as.jpa.spi**
- **org.jboss.as.jpa.util**
- **org.jboss.as.jsr77**
- **org.jboss.as.logging**
- **org.jboss.as.mail**
- **org.jboss.as.management-client-content**
- **org.jboss.as.messaging**
- **org.jboss.as.modcluster**
- **org.jboss.as.naming**

- org.jboss.as.network
- org.jboss.as.osgi
- org.jboss.as.platform-mbean
- org.jboss.as.pojo
- org.jboss.as.process-controller
- org.jboss.as.protocol
- org.jboss.as.remoting
- org.jboss.as.sar
- org.jboss.as.security
- org.jboss.as.server
- org.jboss.as.standalone
- org.jboss.as.threads
- org.jboss.as.transactions
- org.jboss.as.web
- org.jboss.as.webservices
- org.jboss.as.webservices.server.integration
- org.jboss.as.webservices.server.jaxrpc-integration
- org.jboss.as.weld
- org.jboss.as.xts
- org.jboss.classfilewriter
- org.jboss.com.sun.httpserver
- org.jboss.common-core
- org.jboss.dmr
- org.jboss.ejb-client
- org.jboss.ejb3
- org.jboss.iiop-client
- org.jboss.integration.ext-content
- org.jboss.interceptor

- org.jboss.interceptor.spi
- org.jboss.invocation
- org.jboss.ironjacamar.api
- org.jboss.ironjacamar.impl
- org.jboss.ironjacamar.jdbcadapters
- org.jboss.jandex
- org.jboss.jaxbintros
- org.jboss.jboss-transaction-spi
- org.jboss.jsfunit.core
- org.jboss.jts
- org.jboss.jts.integration
- org.jboss.logging
- org.jboss.logmanager
- org.jboss.logmanager.log4j
- org.jboss.marshalling
- org.jboss.marshalling.river
- org.jboss.metadata
- org.jboss.modules
- org.jboss.msc
- org.jboss.netty
- org.jboss.osgi.deployment
- org.jboss.osgi.framework
- org.jboss.osgi.resolver
- org.jboss.osgi.spi
- org.jboss.osgi.vfs
- org.jboss.remoting3
- org.jboss.resteasy.resteasy-atom-provider
- org.jboss.resteasy.resteasy-cdi

- org.jboss.resteasy.resteasy-jackson-provider
- org.jboss.resteasy.resteasy-jaxb-provider
- org.jboss.resteasy.resteasy-jaxrs
- org.jboss.resteasy.resteasy-jsapi
- org.jboss.resteasy.resteasy-multipart-provider
- org.jboss.sasl
- org.jboss.security.negotiation
- org.jboss.security.xacml
- org.jboss.shrinkwrap.core
- org.jboss.staxmapper
- org.jboss.stdio
- org.jboss.threads
- org.jboss.vfs
- org.jboss.weld.api
- org.jboss.weld.core
- org.jboss.weld.spi
- org.jboss.ws.api
- org.jboss.ws.common
- org.jboss.ws.cxf.jbossws-cxf-client
- org.jboss.ws.cxf.jbossws-cxf-factories
- org.jboss.ws.cxf.jbossws-cxf-server
- org.jboss.ws.cxf.jbossws-cxf-transports-httpserver
- org.jboss.ws.jaxws-client
- org.jboss.ws.jaxws-jboss-httpserver-httpspi
- org.jboss.ws.native.jbossws-native-core
- org.jboss.ws.native.jbossws-native-factories
- org.jboss.ws.native.jbossws-native-services
- org.jboss.ws.saaj-impl

- **org.jboss.ws.spi**
- **org.jboss.ws.tools.common**
- **org.jboss.ws.tools.wsconsume**
- **org.jboss.ws.tools.wsprovide**
- **org.jboss.xb**
- **org.jboss.xnio**
- **org.jboss.xnio.nio**
- **org.jboss.xts**
- **org.jdom**
- **org.jgroups**
- **org.joda.time**
- **org.junit**
- **org.omg.api**
- **org.osgi.core**
- **org.picketbox**
- **org.picketlink**
- **org.python.jython.standalone**
- **org.scannotation.scannotation**
- **org.slf4j**
- **org.slf4j.ext**
- **org.slf4j.impl**
- **org.slf4j.jcl-over-slf4j**
- **org.w3c.css.sac**
- **sun.jdk**

バグを報告する

3.7.3. JBoss デプロイメント構造のデプロイメント記述子のリファレンス

このデプロイメント記述子を使用して実行できる主なタスクは次の通りです。

- 明示的なモジュール依存関係を定義する。

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

バグを報告する

第4章 グローバル値

4.1. バルブについて

バルブは、アプリケーションのパイプラインを処理するリクエストに挿入される Java クラスです。バルブはサーブレットフィルターの前にパイプラインへ挿入されます。バルブはリクエストを渡す前に変更を加えることができ、認証またはリクエストのキャンセルなどの他の処理を実行できます。通常、バルブはアプリケーションとパッケージ化されます。

6.1.0 およびそれ以降のバージョンはグローバルバルブをサポートします。

バグを報告する

4.2. グローバルバルブについて

グローバルバルブは、デプロイされたすべてのアプリケーションのパイプラインを処理するリクエストに挿入されるバルブです。バルブは JBoss Enterprise Application Platform 6 の静的モジュールとしてパッケージ化およびインストールされ、グローバルバルブとなります。グローバルバルブは Web サブシステムで設定されます。

6.1.0 およびそれ以降のバージョンのみがグローバルバルブをサポートします。

バグを報告する

4.3. オーセンティケーターバルブについて

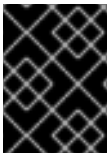
オーセンティケーターバルブは、リクエストの証明情報を認証するバルブです。オーセンティケーターバルブは **`org.apache.catalina.authenticator.AuthenticatorBase`** のサブクラスで、**`authenticate()`** メソッドを上書きします。

このバルブを使用して追加の認証スキームを実装できます。

バグを報告する

4.4. WEB アプリケーションがバルブを使用するよう設定する

グローバルバルブとしてインストールされないバルブは、アプリケーションに含め、**`jboss-web.xml`** デプロイメント記述子で設定する必要があります。



重要

グローバルバルブとしてインストールされたバルブは、デプロイされたすべてのアプリケーションに自動的に適用されます。

要件

- バルブを作成し、アプリケーションのクラスパスに含める必要があります。これは、バルブをアプリケーションの WAR ファイルにまたは依存関係として追加されたモジュールに含めることにより、実現できます。このようなモジュールの例には、サーバーにインストールされた静的モジュールや EAR アーカイブの **`lib/`** ディレクトリーにある JAR ファイル (WAR が EAR でデプロイされる場合) があります。
- アプリケーションは **`jboss-web.xml`** デプロイメント記述子を含む必要があります。

手順4.1 ローカルバルブのためにアプリケーションを設定する

1. **バルブ要素を追加する**
   name と class-name の属性を使用してバルブ要素をアプリケーションの **jboss-web.xml** ファイルに追加します。name は、バルブの一意の ID であり、class-name はバルブクラスの名前です。

```
<valve name="VALVENAME" class-name="VALVECLASSNAME">

</valve>
```

2. **特定のパラメーター**
   バルブでパラメーターを設定できる場合は、各パラメーターのバルブ要素に **param** 子要素を追加してそれぞれに対して名前と値を指定します。

```
<param name="PARAMNAME" value = "VALUE" />
```

アプリケーションがデプロイされた場合、バルブは、指定された設定でアプリケーションに対して有効になります。

例4.1 jboss-web.xml バルブ設定

```
<valve name="clientlimiter" class-
name="org.jboss.samplevalves.restrictedUserAgentsValve">
    <param name="restricteduseragents"  value = "^.*MS Web Services
Client Protocol.*$" />
</valve>
```

バグを報告する

4.5. WEB アプリケーションがオーセンティケーターバルブを使用するよう設定する

アプリケーションがオーセンティケーターバルブを使用するよう設定するには、バルブをインストールおよび設定し (アプリケーションに対してローカル、またはグローバルバルブとして)、アプリケーションの **web.xml** デプロイメント記述子を設定する必要があります。最も単純なケースでは、**web.xml** 設定は **BASIC** 認証を使用した場合と同じです。ただし、**of login-config** の **auth-method** 子要素は、設定を実行するバルブの名前に設定されます。

要件

- 認証バルブがすでに作成されている必要があります。
- 認証バルブがグローバルバルブの場合、認証バルブはすでにインストールおよび設定されている必要があります。また、設定された名前を知っている必要があります。
- アプリケーションが使用するセキュリティーレルムのレルム名を知っている必要があります。

使用するバルブまたはセキュリティーレルム名を知らない場合は、サーバー管理者に問い合わせてください。

手順4.2 アプリケーションがオーセンティケーターバルブを使用するよう設定する

1. **バルブを設定する**
   ローカルバルブを使用する場合は、**jboss-web.xml** デプロイメント記述子で設定する必要があります。「Web アプリケーションがバルブを使用するよう設定する」を参照してください。

   グローバルバルブを使用する場合、これは不必要です。

2. **セキュリティー設定を web.xml に追加する**
   security-constraint、login-config、security-role などの標準的な要素を使用して、セキュリティー設定をアプリケーションの web.xml ファイルに追加します。login-config 要素で、auth-method の値をオーセンティケーターバルブの名前に設定します。また、realm-name 要素を、アプリケーションが使用している JBoss セキュリティーレルムの名前に設定する必要があります。

```
<login-config>
   <auth-method>VALVE_NAME</auth-method>
   <realm-name>REALM_NAME</realm-name>
</login-config>
```

アプリケーションがデプロイされた場合、要求の認証は設定された認証バルブにより処理されます。

バグを報告する

4.6. カスタムバルブを作成する

バルブは、アプリケーションのサーブレットフィルターの前にアプリケーション用要求処理パイプラインに挿入される Java クラスです。これは、要求を変更または他の動作を実行するために使用できます。このタスクは、バルブを実装するのに必要な基本的な手順を示しています。

手順4.3 カスタムバルブを作成する

1. **バルブクラスを作成する**
   org.apache.catalina.valves.ValveBase のサブクラスを作成します。

```
package org.jboss.samplevalves;

import org.apache.catalina.valves.ValveBase;
import org.apache.catalina.connector.Request;
import org.apache.catalina.connector.Response;

public class restrictedUserAgentsValve extends ValveBase {

}
```

2. **呼び出しメソッドを実装する**
   invoke() メソッドは、このバルブがパイプラインで実行されるときに呼び出されます。要求オブジェクトと応答オブジェクトはパラメーターとして渡されます。ここで、要求と応答の処理と変更を行います。

```
public void invoke(Request request, Response response)
{

}
```

3. **次のパイプラインステップを呼び出す**
   呼び出しメソッドが最後に実行する必要があることはパイプラインの次のステップを呼び出し、変更された要求オブジェクトと応答オブジェクトを渡すことです。これは、**`getNext().invoke()`** メソッドを使用して行われます。

```
getNext().invoke(request, response);
```

4. **オプション:パラメーターを指定する**
   バルブを設定可能にする必要がある場合は、パラメーターを追加してこれを有効にします。これは、インスタンス変数と各パラメーターに対するセッターを追加して行います。

```
private String restrictedUserAgents = null;

public void setRestricteduseragents(String mystring)
{
   this.restrictedUserAgents = mystring;
}
```

例4.2 単純なカスタムバルブ

```
package org.jboss.samplevalves;

import java.io.IOException;
import java.util.regex.Pattern;

import javax.servlet.ServletException;
import org.apache.catalina.valves.ValveBase;
import org.apache.catalina.connector.Request;
import org.apache.catalina.connector.Response;

public class restrictedUserAgentsValve extends ValveBase
{
    private String restrictedUserAgents = null;

    public void setRestricteduseragents(String mystring)
    {
        this.restrictedUserAgents = mystring;
    }

    public void invoke(Request request, Response response) throws
IOException, ServletException
    {
      String agent = request.getHeader("User-Agent");
      System.out.println("user-agent: " + agent + " : " +
restrictedUserAgents);
      if (Pattern.matches(restrictedUserAgents, agent))
      {
         System.out.println("user-agent: " + agent + " matches: " +
restrictedUserAgents);
         response.addHeader("Connection", "close");
      }
      getNext().invoke(request, response);
    }
}
```

バグを報告する

第5章 開発者向けのロギング

5.1. はじめに

5.1.1. ロギングについて

ロギングとはアプリケーションから活動に関する記録 (ログ) を受け取り、メッセージ群を記録することです。

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

最新の Java のロギングフレームワークの多くには、正確な時間やメッセージの起源など他の詳細も含まれています。

バグを報告する

5.1.2. JBoss LogManager でサポートされるアプリケーションロギングフレームワーク

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

- JBoss Logging - JBoss Enterprise Application Platform 6 に含まれています。
- Apache Commons Logging - http://commons.apache.org/logging/
- Simple Logging Facade for Java (SLF4J) - http://www.slf4j.org/
- Apache log4j - http://logging.apache.org/log4j/1.2/
- Java SE Logging (java.util.logging) - http://download.oracle.com/javase/6/docs/api/java/util/logging/package-summary.html

バグを報告する

5.1.3. ログレベルについて

ログレベルとは、ログメッセージの性質と重大度を示す列挙値の順序付けされたセットです。 特定のログメッセージのレベルは、そのメッセージを送信するために選択したロギングフレームワークの適切なメソッドを使用して開発者が指定します。

JBoss Enterprise Application Platform 6 はサポートされるアプリケーションロギングフレームワークによって使用されるすべてのログレベルをサポートします。最も一般的に使用される 6 つのログレベルは、ログレベルの低い順に **TRACE**、**DEBUG**、**INFO**、**WARN**、**ERROR** および **FATAL** となります。

ログレベルはログカテゴリとログハンドラーによって使用され、それらが担当するメッセージを限定します。各ログレベルには、他のログレベルに対して相対的な順番を示す数値が割り当てられています。ログカテゴリーとハンドラーにはログレベルが割り当てられ、そのレベル以上のログメッセージのみを処理します。たとえば、**WARN** レベルのログハンドラーは、**WARN**、**ERROR**、および **FATAL** のレベルのメッセージのみを記録します。

バグを報告する

5.1.4. サポートされているログレベル

表5.1 サポートされているログレベル

ログレベル	値	説明
FINEST	300	-
FINER	400	-
TRACE	400	アプリケーションの実行状態に関する詳細情報を提供するメッセージに使用します。通常、**TRACE** のログメッセージはアプリケーションのデバッグ時のみにキャプチャーされます。
DEBUG	500	アプリケーションの個別の要求またはアクティビティの進捗状況を表示するメッセージに使用します。**DEBUG** のログメッセージは通常アプリケーションのデバッグ時のみにキャプチャーされます。
FINE	500	-
CONFIG	700	-
INFO	800	アプリケーションの全体的な進捗状況を示すメッセージに使用します。多くの場合、アプリケーションの起動、シャットダウン、およびその他の主要なライフサイクルイベントに使用されます。
WARN	900	エラーではないが、理想的とは見なされない状況を示すために使用されます。将来的にエラーをもたらす可能性のある状況を示します。
WARNING	900	-
ERROR	1000	発生したエラーの中で、現在のアクティビティや要求の完了を妨げる可能性があるが、アプリケーション実行の妨げにはならないエラーを表示するために使用されます。
SEVERE	1000	-
FATAL	1100	クリティカルなサービス障害やアプリケーションのシャットダウンをもたらしたり、JBoss Enterprise Application Platform 6 のシャットダウンを引き起こす可能性があるイベントを表示するのに使用します。

バグを報告する

5.1.5. デフォルトのログファイルの場所

これらは、デフォルトのロギング設定に対して作成されたログファイルです。デフォルトの設定では、周期的なログハンドラーを使用してサーバーログファイルが書き込まれます。

表5.2 スタンドアローンサーバー用デフォルトログファイル

ログファイル	説明
***EAP_HOME*/standalone/log/boot.log**	サーバールートログ。サーバーの起動に関連するログメッセージが含まれます。

ログファイル	説明
***EAP_HOME*/standalone/log/server.log**	サーバーログ。サーバー起動後のすべてのログメッセージが含まれます。

表5.3 管理対象ドメイン用デフォルトログファイル

ログファイル	説明
***EAP_HOME*/domain/log/host-controller/boot.log**	ホストコントローラーブートログ。ホストコントローラーの起動に関連するログメッセージが含まれます。
***EAP_HOME*/domain/log/process-controller/boot.log**	プロセスコントローラーブートログ。プロセスコントローラーの起動に関連するログメッセージが含まれます。
***EAP_HOME*/domain/servers/*SERVERNAME*/log/boot.log**	指定されたサーバーのサーバーブートログ。指定されたサーバーの起動に関連するログメッセージが含まれます。
***EAP_HOME*/domain/servers/*SERVERNAME*/log/server.log**	指定されたサーバーのサーバーログ。指定されたサーバー起動後のすべてのログメッセージが含まれます。

バグを報告する

5.2. JBOSS ロギングフレームワークを用いたロギング

5.2.1. JBoss Logging について

JBoss ロギングは JBoss Enterprise Application Platform 6 に含まれるアプリケーションロギングフレームワークです。

JBoss ロギングはアプリケーションにロギングを追加する簡単な方法を提供します。フレームワークを使用するアプリケーションにコードを追加し、定義された形式でログメッセージを送信します。アプリケーションサーバーにアプリケーションがデプロイされると、これらのメッセージがサーバーによってキャプチャーされ、サーバーの設定通りファイルに表示されたり書き込まれたりします。

バグを報告する

5.2.2. JBoss ロギングの機能

- 革新的かつ使いやすい「型指定された」ロガーを提供します。
- 国際化および現地化を完全サポートします。翻訳者はプロパティーファイルのメッセージバンドルを使用します。開発者はインターフェースやアノテーションを使用できます。
- 実稼働向けにはビルド時に型指定されたロガーを生成し、開発向けにはランタイムで型指定されたロガーを生成するツール。

バグを報告する

5.2.3. JBoss ロギングを使用してアプリケーションにロギングを追加

アプリケーションからのメッセージをログに記録するために、Logger オブジェクト (**org.jboss.logging.Logger**) を作成し、そのオブジェクトの適切なメソッドを呼び出します。このタスクは、アプリケーションにこのオブジェクトのサポートを追加するために必要な手順を示しています。

要件

このタスクを行う前に、 次の条件を満たす必要があります。

- ビルドシステムとして Maven を使用している場合は、JBoss Maven リポジトリーを含めるようプロジェクトが設定されている必要があります。「Maven 設定を使用した JBoss Enterprise Application Platform の Maven リポジトリーの設定」を参照してください。
- JBoss ロギング JAR ファイルがアプリケーションのビルドパスに指定されている必要があります。これを行う方法は、アプリケーションをビルドするのに JBoss Developer Studio を使用するか、Maven を使用するかによって異なります。
  - JBoss Developer Studio を使用してビルドする場合、これを行うには JBoss Developer Studio メニューから **[Project] -> [Properties]** を選択し、**[Targeted Runtimes]** を選択して、JBoss Enterprise Application Platform 6 のランタイムがチェックされていることを確認します。
  - Maven を使用してビルドする場合、これを行うには、次の依存性設定をプロジェクトの **pom.xml** ファイルに追加します。

    ```
    <dependency>
       <groupId>org.jboss.logging</groupId>
       <artifactId>jboss-logging</artifactId>
       <version>3.1.2.GA-redhat-1</version>
       <scope>provided</scope>
    </dependency>
    ```

    JAR は、JBoss Enterprise Application Platform 6 がデプロイされたアプリケーションに提供するため、ビルドされたアプリケーションに含める必要はありません。

プロジェクトが正しくセットアップされたら、ロギングを追加する各クラスに対して次の手順を実行する必要があります。

1. **インポートの追加**
   使用する JBoss Logging クラスネームスペースに対して import ステートメントを追加します。少なくとも、**import org.jboss.logging.Logger** をインポートする必要があります。

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

2. **Logger オブジェクトの作成**
   org.jboss.logging.Logger のインスタンスを作成し、静的メソッド **Logger.getLogger(Class)** を呼び出して初期化します。各クラスに対してこれを単一インスタンス変数として作成することが推奨されます。

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

3. **ロギングメッセージの追加**
   Logger オブジェクトのメソッドへの呼び出しを、ログメッセージを送信するコードに追加します。**Logger** オブジェクトには、さまざまなタイプのメッセージに対するさまざまなパラメーターを持つさまざまなメソッドが含まれます。最も使用しやすいものは次のとおりです。

debug(Object message)
info(Object message)
error(Object message)
trace(Object message)
fatal(Object message)

   これらのメッセージは、対応するログレベルと **message** パラメーターを文字列として持つログメッセージを送信します。

```
LOGGER.error("Configuration file not found.");
```

   JBoss Logging メソッドの完全なリストについては、JBoss Enterprise Application Platform 6 API ドキュメンテーションの **org.jboss.logging** パッケージを参照してください。

例5.1 プロパティーファイルを開くときに JBoss ロギングを使用

次の例は、プロパティーファイルからアプリケーションのカスタマイズされた設定をロードするクラスのコードの一部を示しています。指定されたファイルが見つからない場合は、エラーレベルログメッセージが記録されます。

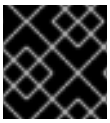
```
import org.jboss.logging.Logger;
public class LocalSystemConfig
{
   private static final Logger LOGGER =
Logger.getLogger(LocalSystemConfig.class);

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

```
            throw new CustomConfigFileNotFoundException(configname);
        }

        return props;
    }
```

バグを報告する

5.3. ロギングプロファイル

5.3.1. ロギングプロファイルについて



重要

ロギングプロファイルは 6.1.0 およびそれ以降のバージョンでのみ使用可能です。

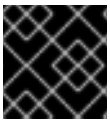
ロギングプロファイルは、デプロイされたアプリケーションに割り当てられる独立したロギング設定のセットです。ロギングプロファイルはハンドラー、カテゴリーおよびルートロガーを通常のロギングサブシステム同様に定義できますが、他のプロファイルやメインのロギングサブシステムを参照できません。ロギングプロファイルは設定を容易にするためロギングサブシステムに似ています。

ログインプロファイルを使用すると、管理者は他のロギング設定に影響を与えることなく 1 つ以上のアプリケーションに固有するロギング設定を作成することができます。各プロファイルはサーバー設定に定義されるため、影響するアプリケーションを再デプロイする必要はなく、ロギング設定を変更できます。

各ロギングプロファイルに含めることができる設定は次のとおりです。

- 一意の名前 (必須）。
- ログハンドラーの数 (いくつでも)。
- ログカテゴリーの数 (いくつでも)。
- ルートロガー (1 つまで)。

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



重要

ロギングプロファイルは管理コンソールを使用して設定できません。

バグを報告する

5.3.2. アプリケーションにおけるロギングプロファイルの指定

アプリケーションは使用するロギングプロファイルを **MANIFEST.MF** ファイルに指定します。

前提条件

1. サーバー上に設定されたロギングプロファイルの名前を認識している必要があります。使用するプロファイルの名前についてはサーバー管理者に問い合わせてください。

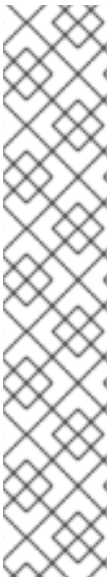
手順5.1 ロギングプロファイル設定をアプリケーションへ追加

- **MANIFEST.MF の編集**
  アプリケーションに **MANIFEST.MF** ファイルがない場合は、以下の内容が含まれるファイルを作成します。*NAME* は必要なプロファイル名に置き換えてください。

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

  アプリケーションに **MANIFEST.MF** ファイルがある場合は、以下の行を追加し、*NAME* を必要なプロファイル名に置き換えます。

```
Logging-Profile: NAME
```



注記

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

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

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

バグを報告する

第6章 国際化と現地語化

6.1. はじめに

6.1.1. 国際化について

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

バグを報告する

6.1.2. 多言語化について

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

バグを報告する

6.2. JBOSS ロギングツール

6.2.1. 概要

6.2.1.1. JBoss ロギングツールの国際化および現地語化

JBoss ロギングツールは、ログメッセージや例外メッセージ、汎用文字列などの国際化や現地語化のサポートを提供する Java API です。JBoss ロギングツールは翻訳のメカニズムを提供するだけでなく、各ログメッセージに対して一意な識別子のサポートも提供します。

国際化されたメッセージや例外は、`org.jboss.logging` アノテーションが付けられたインターフェース内でメソッド定義として作成されます。JBoss ロギングがコンパイル時にインターフェースを実装するため、インターフェースを実装する必要はありません。定義すると、これらのメソッドを使用してコードでメッセージをログに記録したり、例外オブジェクトを取得することが可能です。

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

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

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

バグを報告する

6.2.1.2. JBoss ロギングツールのクイックスタート

JBoss ロギングツールのクイックスタート **logging-tools** には JBoss ロギングツールの機能を実証する単純な Maven プロジェクトが含まれています。このクイックスタートは本書のコード例で幅広く使用されています。

このクイックスタートを参照すると、本書で説明されている全機能を完全実証することができます。

バグを報告する

6.2.1.3. メッセージロガー

メッセージロガーは国際化されたログメッセージを定義するために使用されるインターフェースです。メッセージロガーには **@org.jboss.logging.MessageLogger** アノテーションが付けられます。

バグを報告する

6.2.1.4. メッセージバンドル

メッセージバンドルは、汎用の翻訳可能なメッセージや国際化されたメッセージが含まれる例外オブジェクトの定義に使用できるインターフェースです。メッセージバンドルはログメッセージの作成には使用されません。

メッセージバンドルインターフェースには **@org.jboss.logging.MessageBundle** アノテーションが付けられます。

バグを報告する

6.2.1.5. 国際化されたログメッセージ

国際化されたログメッセージは、メッセージロガーのメソッドで定義を行い作成されるログメッセージです。メソッドには **@LogMessage** と **@Message** アノテーションを付ける必要があり、**@Message** の値属性を使用してログメッセージを指定しなければなりません。国際化されたログメッセージはプロパティーファイルに翻訳を提供するとローカライズされます。

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

バグを報告する

6.2.1.6. 国際化された例外

国際化された例外はメッセージバンドルで定義されたメソッドから返された例外オブジェクトです。Java Exception オブジェクトを返すメッセージバンドルメソッドにアノテーションを付けてデフォルトの例外メッセージを定義することができます。現在のロケールに一致するプロパティーファイルに翻訳があると、デフォルトメッセージは翻訳に置き換えられます。国際化された例外にもプロジェクトコードとメッセージ ID が割り当てられています。

バグを報告する

6.2.1.7. 国際化されたメッセージ

国際化されたメッセージはメッセージバンドルに定義されるメソッドから返された文字列です。Java String オブジェクトを返すメッセージバンドルメソッドにアノテーションを付け、その文字列のデフォルトの内容 (メッセージ) を定義することができます。現在のロケールに一致するプロパティーファイルに翻訳があると、デフォルトメッセージは翻訳に置き換えられます。

バグを報告する

6.2.1.8. 翻訳プロパティーファイル

翻訳プロパティーファイルは、1 つのロケール、国、バリアントに対する 1 つのインターフェースからのメッセージ翻訳が含まれる Java プロパティーファイルです。翻訳プロパティーファイルは、メッセージを返すクラスを生成するため JBoss ロギングツールによって使用されます。

バグを報告する

6.2.1.9. JBoss ロギングツールのプロジェクトコード

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

バグを報告する

6.2.1.10. JBoss ロギングツールのメッセージ ID

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

バグを報告する

6.2.2. 国際化されたロガー、メッセージ、例外の作成

6.2.2.1. 国際化されたログメッセージの作成

このタスクでは、JBoss ロギングツールを使用して MessageLogger インターフェースを作成することにより、国際化されたログメッセージを作成する方法を示します。すべてのオプション機能またはログメッセージの国際化については取り上げません。

完全な例は `logging-tools` クイックスタートを参照してください。

前提条件

1. Maven プロジェクトがすでに存在している必要があります。「JBoss ロギングツールの Maven 設定」を参照してください。
2. JBoss ロギングツールに必要な Maven 設定がプロジェクトにある必要があります。

手順6.1 国際化されたログメッセージバンドルの作成

1. **メッセージロガーインターフェースの作成**
   ログメッセージの定義が含まれるように Java インターフェースをプロジェクトに追加します。定義されるログメッセージに対し、インターフェースにその内容を表す名前を付けます。

   ログメッセージインターフェースの要件は次の通りです。

   - `@org.jboss.logging.MessageLogger` アノテーションが付けられていなければなりません。

- **org.jboss.logging.BasicLogger** を拡張しなければなりません。
- このインターフェースを実装する型付きロガーのフィールドをインターフェースが定義する必要があります。**org.jboss.logging.Logger** の **getMessageLogger()** メソッドで定義します。

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

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

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

2. **メソッド定義の追加**
   各ログメッセージのインターフェースにメソッド定義を追加します。ログメッセージに対する各メソッドにその内容を表す名前を付けます。

   各メソッドの要件は次の通りです。

   - メソッドは **void** を返さなければなりません。
   - **@org.jboss.logging.LogMessage** アノテーションが付いていなければなりません。
   - **@org.jboss.logging.Message** アノテーションが付いていなければなりません。
   - **@org.jboss.logging.Message** の値属性にはデフォルトのログインメッセージが含まれます。翻訳がない場合にこのメッセージが使用されます。

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

   デフォルトのログレベルは **INFO** です。

3. **メソッドの呼び出し**
   メッセージがロギングされなければならない場所にコードのインターフェースメソッドへの呼び出しを追加します。プロジェクトがコンパイルされる時にアノテーションプロセッサーがインターフェースの実装を作成するため、インターフェースの実装を作成する必要はありません。

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

   カスタムのロガーは BasicLogger よりサブクラス化されるため、**BasicLogger** のロギングメソッド (**debug()** や **error()** など) を使用することもできます。国際化されていないメッセージをログに記録するため他のロガーを作成する必要はありません。

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

結果: 現地語化できる 1 つ以上の国際化されたロガーをプロジェクトがサポートするようになります。

バグを報告する

6.2.2.2. 国際化されたメッセージの作成と使用

このタスクでは、国際化されたメッセージの作成方法と使用方法を示します。すべてのオプション機能またはメッセージの国際化プロセスについては取り上げません。

完全な例は **logging-tools** クイックスタートを参照してください。

要件

1. JBoss Enterprise Application Platform 6 のリポジトリを使用する作業用の Maven プロジェクトが存在しなければなりません。「Maven 設定を使用した JBoss Enterprise Application Platform の Maven リポジトリーの設定」 を参照してください。
2. JBoss ロギングツールの必要な Maven 設定が追加されている必要があります。「JBoss ロギングツールの Maven 設定」 を参照してください。

手順6.2 国際化されたメッセージの作成と使用

1. **例外のインターフェースの作成**
   JBoss ロギングツールはインターフェースで国際化されたメッセージを定義します。定義されるメッセージに対し、インターフェースにその内容を表す名前を付けます。

   インターフェースの要件は次の通りです。

   - パブリックとして宣言される必要があります。
   - **@org.jboss.logging.MessageBundle** アノテーションが付けられていなければなりません。
   - インターフェースと同じ型のメッセージバンドルであるフィールドをインターフェースが定義する必要があります。

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

2. **メソッド定義の追加**
   各メッセージのインターフェースにメソッド定義を追加します。メッセージに対する各メソッドにその内容を表す名前を付けます。

   各メソッドの要件は次の通りです。

   - 型 **String** のオブジェクトを返す必要があります。
   - **@org.jboss.logging.Message** アノテーションが付いていなければなりません。
   - デフォルトメッセージに **@org.jboss.logging.Message** の値属性が設定されていなければなりません。翻訳がない場合にこのメッセージが使用されます。

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

3. **呼び出しメソッド**
   メッセージを取得する必要がある場所でアプリケーションのインターフェースメソッドを呼び出します。

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

結果: 現地語化できる国際化されたメッセージをプロジェクトがサポートするようになります。

バグを報告する

6.2.2.3. 国際化された例外の作成

このタスクでは、国際化された例外の作成方法と使用方法を示します。すべてのオプション機能またはこれらの例外の国際化プロセスについては取り上げません。

完全な例は **logging-tools** クイックスタートを参照してください。

このタスクでは、 JBoss Developer Studio または Maven に構築されたソフトウェアプロジェクトが既に存在し、このプロジェクトに国際化された例外を追加することを前提としています。

手順6.3 国際化された例外の作成と使用

1. **JBoss ロギングツール設定の追加**
   JBoss ロギングツールをサポートするために必要なプロジェクト設定を追加します。「JBoss ロギングツールの Maven 設定」を参照してください。

2. **例外のインターフェースの作成**
   JBoss ロギングツールはインターフェースで国際化された例外を定義します。定義される例外に対し、インターフェースにその内容を表す名前を付けます。

   インターフェースの要件は次の通りです。

   - **public** として宣言される必要があります。
   - **@org.jboss.logging.MessageBundle** アノテーションが付けられていなければなりません。
   - インターフェースと同じ型のメッセージバンドルであるフィールドをインターフェースが定義する必要があります。

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

3. **メソッド定義の追加**
   各例外のインターフェースにメソッド定義を追加します。例外に対する各メソッドにその内容を表す名前を付けます。

各メソッドの要件は次の通りです。

- 型 **Exception** のオブジェクトまたは **Exception** のサブタイプを返す必要があります。
- **@org.jboss.logging.Message** アノテーションが付いていなければなりません。
- デフォルトの例外メッセージに **@org.jboss.logging.Message** の値属性が設定されていなければなりません。翻訳がない場合にこのメッセージが使用されます。
- メッセージ文字列の他にパラメータを必要とするコンストラクターが返された例外にある場合、**@Param** アノテーションを使用してこれらのパラメーターをメソッド定義に提供しなければなりません。パラメーターはコンストラクターと同じ型で同じ順番でなければなりません。

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

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

4. **呼び出しメソッド**
   例外を取得する必要がある場所でコードのインターフェースメソッドを呼び出します。メソッドは例外をスローしませんが、スローできる例外オブジェクトを返します。

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

結果: 現地語化できる国際化された例外をプロジェクトがサポートするようになります。

バグを報告する

6.2.3. 国際化されたロガー、メッセージ、例外の現地語化

6.2.3.1. Maven で新しい翻訳プロパティーファイルを生成する

Maven で構築されたプロジェクトは、各メッセージロガーに対する空の翻訳プロパティーファイルと含まれるメッセージバンドルを生成できます。これらのファイルは新しい翻訳ファイルとして使用することができます。

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

完全な例は **logging-tools** クイックスタートを参照してください。

前提条件

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

手順6.4 Maven で新しい翻訳プロパティーファイルを生成する

1. **Maven 設定の追加**
   -AgenereatedTranslationFilePath コンパイラー引数を Maven コンパイラープラグイン設定に追加し、新しいファイルが作成されるパスを割り当てます。

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

   上記の設定は Maven プロジェクトの **target/generated-translation-files** ディレクトリに新しいファイルを作成します。

2. **プロジェクトの構築**
   Maven を使用したプロジェクトの構築

```
[Localhost]$ mvn compile
```

@MessageBundle または **@MessageLogger** アノテーションが付けられたインターフェースごとに 1 つのプロパティーファイルが作成されます。各インターフェースが宣言される Java パッケージに対応するサブディレクトリに新しいファイルが作成されます。

各ファイルは、***InterfaceName*.i18n_locale_COUNTRY_VARIANT.properties** という構文を使用して名前が付けられます。**InterfaceName** はこのファイルが生成されたインターフェースの名前になります。

新しい翻訳の基盤としてこれらのファイルをプロジェクトへコピーすることができます。

バグを報告する

6.2.3.2. 国際化されたロガーや例外、メッセージの翻訳

JBoss ロギングツールを使用してインターフェースに定義されたロギングメッセージや例外メッセージの翻訳をプロパティーファイルに提供することが可能です。

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

完全な例は **logging-tools** クイックスタートを参照してください。

要件

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

手順6.5 国際化されたロガーや例外、メッセージの翻訳

1. **テンプレートプロパティーファイルの生成**
   mvn compile コマンドを実行し、テンプレート翻訳プロパティーファイルを作成します。
2. **プロジェクトへのテンプレートファイルの追加**
   翻訳したいインターフェースのテンプレートを、テンプレートが作成されたディレクトリからプロジェクトの **src/main/resources** ディレクトリへコピーします。プロパティーファイルは翻訳するインターフェースと同じパッケージに存在しなければなりません。
3. **コピーしたテンプレートファイルの名前変更**
   GreeterLogger.i18n_fr_FR.properties のように、含まれる翻訳に応じてテンプレートファイルのコピーの名前を変更します。
4. **テンプレートの内容の翻訳**
   新しい翻訳プロパティーファイルを編集し、適切な翻訳が含まれるようにします。

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

   実行された各バンドルの各翻訳に対して手順の 2、3、4 を繰り返します。

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

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

バグを報告する

6.2.4. 国際化されたログメッセージのカスタマイズ

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

このタスクではメッセージ ID とプロジェクトコードを国際化されたログメッセージへ追加する方法を説明します。ログメッセージがログで表示されるようにするには、プロジェクトコードとメッセージ ID の両方が必要となります。メッセージにプロジェクトコードとメッセージ ID の両方がない場合、どちらも表示されません。

完全な例は **`logging-tools`** クイックスタートを参照してください。

要件

1. 国際化されたログメッセージを持つプロジェクトが存在する必要があります。「国際化されたログメッセージの作成」を参照してください。
2. 使用するプロジェクトコードを認識する必要があります。プロジェクトコードを 1 つ使用することも、各インターフェースに異なるコードを定義することも可能です。

手順6.6 メッセージ ID とプロジェクトコードをログメッセージに追加する

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

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

}
```

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

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

メッセージ ID とプロジェクトコードの両方が関連付けられたログメッセージは、メッセージ IDとプロジェクトコードをログに記録されたメッセージの前に付けます。

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

バグを報告する

6.2.4.2. メッセージのログレベル設定

JBoss ロギングツールのインターフェースによって定義されるメッセージのログレベルのデフォルトは **`INFO`** です。ロギングメソッドに付けられた**`@LogMessage`** アノテーションの **`level`** 属性を用いて異なるログレベルを指定することが可能です。

手順6.7 メッセージのログレベルの指定

1. **level 属性の指定**
   ログメッセージメソッド定義の **@LogMessage** アノテーションに **level** 属性を追加します。

2. **ログレベルの割り当て**
   このメッセージに対するログレベルの値を **level** 属性に割り当てます。**level** に有効な値は **org.jboss.logging.Logger.Level** に定義される 6 つの列挙定数である **DEBUG**、**ERROR**、**FATAL**、 **INFO**、**TRACE**、 **WARN** になります。

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

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

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

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

バグを報告する

6.2.4.3. パラメーターによるログメッセージのカスタマイズ

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

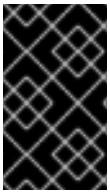
手順6.8 パラメーターによるログメッセージのカスタマイズ

1. **メソッド定義へパラメーターを追加する**
   すべての型のパラメーターをメソッド定義に追加することができます。型に関係なくパラメーターの String 表現がメッセージに表示されます。

2. **ログメッセージへパラメーター参照を追加する**
   参照は明示的なインデックスか通常のインデックスを使用できます。

   - 通常のインデックスを使用するには、各パラメーターを表示したいメッセージ文字列に **%s** 文字を挿入します。**%s** の最初のインスタンスにより最初のパラメーターが挿入され、2 番目のインスタンスにより 2 番目のパラメーターが挿入されます。

   - 明示的なインデックスを使用するには、文字 **%{#}** をメッセージに挿入します。**#** は表示したいパラメーターの数に置き換えます。



重要

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

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

例6.1 通常のインデックスを使用したメッセージパラメーター

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

例6.2 明示的なインデックスを使用したメッセージパラメーター

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

バグを報告する

6.2.4.4. ログメッセージの原因として例外を指定する

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

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

手順6.9 ログメッセージの原因として例外を指定する

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

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

2. **アノテーションの追加**
   パラメーターに **@Cause** アノテーションを追加します。

   ```
   import org.jboss.logging.Cause

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

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

   ```
   try
   {
   ```

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

コードが **FileNotFoundException** 型の例外をスローした場合、上記コード例の出力は次のようになります。

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

バグを報告する

6.2.5. 国際化された例外のカスタマイズ

6.2.5.1. メッセージ ID とプロジェクトコードを例外メッセージに追加する

以下の手順は、JBoss ロギングツールを使用して作成された国際化済み例外メッセージにメッセージ ID とプロジェクトコードを追加するために必要な作業を示します。

メッセージ ID とプロジェクトコードは国際化された例外によって表示された各メッセージの前に付けられる固有の識別子です。これらの識別コードによってアプリケーションに対する全例外メッセージの参照を作成できるため、理解できない言語で書かれた例外メッセージの意味をルックアップすることが可能です。

要件

1. 国際化された例外を持つプロジェクトが存在する必要があります。「国際化された例外の作成」を参照してください。
2. 使用するプロジェクトコードを認識する必要があります。プロジェクトコードを 1 つ使用することも、各インターフェースに異なるコードを定義することも可能です。

手順6.10 メッセージ ID とプロジェクトコードを例外メッセージに追加する

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

```
@MessageBundle(projectCode="ACCTS")
interface ExceptionBundle
{
```

```
   ExceptionBundle EXCEPTIONS =
Messages.getBundle(ExceptionBundle.class);
}
```

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

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

重要

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

例6.3 国際化された例外の作成

この例外バンドルインターフェースは、プロジェクトコード ACCTS と ID が 143 の例外メソッドを 1 つ持っています。

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

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

次のコードを使用して例外オブジェクトを取得したりスローしたりすることが可能です。

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

これにより、次のような例外メッセージが表示されます。

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

バグを報告する

6.2.5.2. パラメーターによる例外メッセージのカスタマイズ

例外を定義する例外バンドルメソッドは、パラメーターを指定して例外メッセージに表示される追加情報を渡すことが可能です。例外メッセージでパラメーターが表示される場所は、明示的なインデクシングか通常のインデクシングを使用してメッセージ自体に指定されます。

以下の手順では、メソッドパラメーターを使用してメソッド例外をカスタマイズするために必要な作業について説明します。

手順6.11 パラメーターによる例外メッセージのカスタマイズ

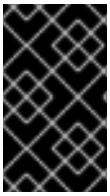
1. **メソッド定義へパラメーターを追加する**
   すべての型のパラメーターをメソッド定義に追加することができます。型に関係なくパラメーターの `String` 表現がメッセージに表示されます。

2. **例外メッセージへパラメーター参照を追加する**
   参照は明示的なインデックスか通常のインデックスを使用できます。

   - 通常のインデックスを使用するには、各パラメーターを表示したいメッセージ文字列に **%s** 文字を挿入します。**%s** の最初のインスタンスにより最初のパラメーターが挿入され、2 番目のインスタンスにより 2 番目のパラメーターが挿入されます。

   - 明示的なインデックスを使用するには、文字 **%{#}** をメッセージに挿入します。# は表示したいパラメーターの数に置き換えます。

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



重要

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

例6.4 通常のインデックスを使用

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

例6.5 明示的なインデックスを使用

```
@Message(id=143, value = "The config file %{1} could not be opened.")
IOException configFileAccessError(File config);
```

バグを報告する

6.2.5.3. 別の例外の原因として 1 つの例外を指定する

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

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

手順6.12 別の例外の原因として 1 つの例外を指定する

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

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

2. **アノテーションの追加**
   パラメーターに **@Cause** アノテーションを追加します。

```
import org.jboss.logging.Cause

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

3. **メソッドの呼び出し**
   例外オブジェクトを取得するためインターフェースメソッドを呼び出します。キャッチした例外を原因として使用し、キャッチブロックより新しい例外をスローするのが最も一般的なユースケースになります。

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

例6.6 別の例外の原因として 1 つの例外を指定する

この例外バンドルは、**ArithmeticException** 型の例外を返す単一のメソッドを定義します。

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

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

}
```

このコードスニペットは、整数のゼロ除算を実行しようとすると例外をスローする演算を行います。最初の例外を原因として使用して例外がキャッチされ、新しい例外が作成されます。

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

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

例外メッセージは次のようになります。

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

バグを報告する

6.2.6. 参考資料

6.2.6.1. JBoss ロギングツールの Maven 設定

国際化に JBoss ロギングツールを使用する Maven プロジェクトを構築するには、**pom.xml** ファイルのプロジェクトの設定を次のように変更する必要があります。

完全な **pom.xml** ファイルの例については、**logging-tools** クイックスタートを参照してください。

1. プロジェクトに対して JBoss Maven リポジトリが有効になっている必要があります。「Maven 設定を使用した JBoss Enterprise Application Platform の Maven リポジトリーの設定」を参照してください。
2. **jboss-logging** と **jboss-logging-processor** の Maven 依存関係を追加する必要があります。これらの依存関係は両方 JBoss Enterprise Application Platform 6 で使用可能であるため、各依存関係の **scope** 要素を次のように **provided** に設定できます。

```
<dependency>
   <groupId>org.jboss.logging</groupId>
   <artifactId>jboss-logging-processor</artifactId>
   <version>1.0.0.Final</version>
   <scope>provided</scope>
</dependency>
```

```
<dependency>
   <groupId>org.jboss.logging</groupId>
   <artifactId>jboss-logging</artifactId>
   <version>3.1.0.GA</version>
   <scope>provided</scope>
</dependency>
```

3. **maven-compiler-plugin** のバージョンは **2.2** 以上である必要があり、**1.6** のターゲットソースおよび生成されたソースに対して設定する必要があります。

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>2.3.2</version>
   <configuration>
      <source>1.6</source>
      <target>1.6</target>
   </configuration>
</plugin>
```

バグを報告する

6.2.6.2. 翻訳プロパティーファイルの形式

JBoss ロギングツールでのメッセージの翻訳に使用されるプロパティーファイルは標準的な Java プロパティーファイルです。このファイルの形式は、http://docs.oracle.com/javase/6/docs/api/java/util/Properties.html の **java.util.Properties** クラスのドキュメントに記載されている単純な行指向の**key=value** ペア形式です。

ファイル名の形式は次のようになります。

```
InterfaceName.i18n_locale_COUNTRY_VARIANT.properties
```

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

翻訳ファイルに含まれるプロパティーは翻訳されたインターフェースからのメソッド名です。プロパティに割り当てられた値が翻訳になります。メソッドがオーバロードされると、ドットと名前へのパラメーター数が付加されます。翻訳のメソッドは異なるパラメーター数が提供される場合のみオーバーロードされます。

例6.7 翻訳プロパティーファイルの例

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

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

バグを報告する

6.2.6.3. JBoss ロギングツールのアノテーションに関する参考資料

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

表6.1 JBoss ロギングツールのアノテーション

アノテーション	ターゲット	説明	属性
@MessageBundle	インターフェース	インターフェースをメッセージバンドルとして定義します。	**projectCode**
@MessageLogger	インターフェース	インターフェースをメッセージロガーとして定義します。	**projectCode**
@Message	メソッド	メッセージバンドルとメッセージロガーで使用できます。メッセージロガーでは、多言語化されたロガーとしてメソッドを定義します。メッセージバンドルでは、多言語化された文字列または例外オブジェクトを返すメソッドとして定義します。	**value**、**id**
@LogMessage	メソッド	メッセージロガーのメソッドをロギングメソッドとして定義します。	**level**(デフォルトは **INFO**)
@Cause	パラメーター	ログメッセージまたは他の例外が発生したときに例外を渡すパラメーターとして定義します。	-
@Param	パラメーター	例外のコンストラクターへ渡されるパラメーターとして定義します。	-

バグを報告する

第7章 ENTERPRISE JAVABEANS

7.1. はじめに

7.1.1. Enterprise JavaBeans の概要

Enterprise JavaBeans (EJB) 3.1 は、Enterprise Bean と呼ばれるサーバーサイドコンポーネントを使用してセキュアでポータブルな分散 Java EE アプリケーションを開発するための API です。Enterprise Bean は、再利用を促進する分離された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。Enterprise JavaBeans 3.1 は、Java EE 仕様 JSR-318 としてドキュメント化されています。

JBoss Enterprise Application Platform 6 では、Enterprise JavaBeans 3.1 仕様を使用してビルドされたアプリケーションが完全にサポートされます。EJB コンテナは JBoss EJB3 コミュニティープロジェクト (http://www.jboss.org/ejb3) を使用して実装されます。

バグを報告する

7.1.2. EJB 3.1 機能セット

以下の機能が EJB 3.1 でサポートされています。

- セッション Bean
- メッセージ駆動型 Bean
- ノーインターフェースビュー (No-interface view)
- ローカルインターフェース
- リモートインターフェース
- JAX-WS web サービス
- JAX-RS web サービス
- 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 相互運用性
- トランザクションサポート
- セキュリティ
- 埋め込み API

以下の機能は EJB 3.1 で対応していますが、「プルーニング」用として提案されています。そのため、これらの機能は Java EE 7 ではオプションとなる可能性があります。

- エンティティー Bean (コンテナおよび Bean 管理の永続性)
- EJB 2.1 エンティティー Bean のクライアントビュー

- EJB クエリ言語 (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)

バグを報告する

7.1.3. EJB 3.1 Lite

EJB Lite は EJB 3.1 仕様のサブセットであり、Java EE 6 Web プロファイルの一部として完全な EJB 3.1 仕様の単純なバージョンを提供します。

EJB Lite により、エンタープライズを使用すると、エンタープライズ Bean を使用した Web アプリケーションでのビジネスロジックの実装が簡略化されます。

1. Web アプリケーションに適切な機能のみをサポートします。
2. EJB を同じ WAR ファイルで Web アプリケーションとしてデプロイできます。

バグを報告する

7.1.4. EJB 3.1 Lite の機能

EJB Lite には、次の機能があります。

- ステートレス、ステートフル、およびシングルトンセッション Bean
- ローカルビジネスインターフェースおよび「インターフェースなし」Bean
- インターセプター
- コンテナ管理および Bean 管理トランザクション
- 宣言およびプログラミング可能なセキュリティ
- 埋め込み API

EJB 3.1 の次の機能は含まれていません。

- リモートインターフェース
- RMI と IIOP の相互運用性
- JAX-WS Web サービスエンドポイント
- EJB タイマーサービス
- 非同期セッション Bean 呼び出し
- メッセージ駆動 Bean

バグを報告する

7.1.5. エンタープライズ Bean

Enterprise JavaBeans (EJB) 3.1 仕様、JSR-318 に定義されているように、エンタープライズ Bean はサーバー側のアプリケーションコンポーネントのことです。エンタープライズ Bean は疎結合方式でアプリケーションのビジネスロジックを実装し再利用ができるように設計されています。

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。アプリケーションサーバーに独自のアーカイブ (JAR ファイル) でデプロイするか、 Java EE アプリケーションの一部としてデプロイすることが可能です。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、セキュリティーやトランザクション、同時処理制御などのサービスを提供します。

エンタープライズ Bean はビジネスインターフェースをいくつでも定義することができます。ビジネスインターフェースは、クライアントが使用できる Bean のメソッドに対して優れた制御機能を提供し、リモート JVM で実行され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、メッセージ駆動型 Bean、およびエンティティー Bean の 3 種類があります。

重要

エンティティー Bean は EJB 3.1 で廃止されました。Red Hat は代わりに JPA エンティティーの使用を推奨します。Red Hat はレガシーシステムで後方互換性に対応する場合のみエンティティー Bean の使用を推奨します。

バグを報告する

7.1.6. エンタープライズ Bean の記述について

エンタープライズ Bean はサーバー側のコンポーネントで、特定のアプリケーションクライアントから分離された状態でビジネスロジックをカプセル化するためのものです。エンタープライズ Bean 内にビジネスロジックを実装すると、これらの Bean を複数のアプリケーションで再使用することができます。

エンタープライズ Bean はアノテーション付けされた Java クラスとして記述されます。特定の EJB インターフェースを実装する必要や、エンタープライズ Bean として考慮される EJB スーパークラスからサブクラス化される必要はありません。

EJB 3.1 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 の JAR ファイルは、アプリケーションサーバーへデプロイしたり、エンタープライズアーカイブ (EAR) ファイルに含まれるようにしてアプリケーションと共にデプロイすることが可能です。また、Bean が EJB 3.1 Lite 仕様に準拠する場合は、エンタープライズ Bean を Web アプリケーションと共に WAR ファイルにデプロイすることも可能です。

バグを報告する

7.1.7. セッション Bean ビジネスインターフェース

7.1.7.1. エンタープライズ Bean のビジネスインターフェース

EJB ビジネスインターフェースは Bean 開発者によって書かれた Java インターフェースで、クライアントが使用できるセッション Bean のパブリックメソッドの宣言を提供します。セッション Bean はゼロ (「インターフェースのない」Bean) を含む、あらゆる数のインターフェースを実装することが可能です。

ビジネスインターフェースをローカルインターフェースまたはリモートインターフェースとして宣言することができますが、両方を宣言することはできません。

バグを報告する

7.1.7.2. EJB ローカルビジネスインターフェース

EJB ローカルビジネスインターフェースは、Bean とクライアントは同じ JVM にある場合に利用可能なメソッドを宣言します。セッション Bean がローカルのビジネスインターフェースを実装する場合、そのインターフェー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。

バグを報告する

7.1.7.3. EJB リモートビジネスインターフェース

EJB リモートビジネスインターフェースは、リモートクライアントで利用可能なメソッドを宣言します。リモートインターフェースを実装するセッション Bean へのリモートアクセスは、自動的に EJB コンテナにより提供されます。

リモートクライアントとは別の JVM で実行するクライアントのことで、別のアプリケーションサーバーにデプロイされている Web アプリケーション、サービス、エンタープライズ Bean、デスクトップなどが含まれます。

ローカルクライアントは、リモートのビジネスインターフェースが公開するメソッドへアクセス可能です。これは、リモートクライアントと同じメソッドを使い実行され、リモートリクエストを出した時に付随する通常のオーバーヘッドがすべて発生します。

バグを報告する

7.1.7.4. EJB のインタフェース以外の Bean

ビジネスインターフェースを実装しないセッション Bean はインターフェース以外の Bean と呼ばれます。インターフェース以外の Bean の公開メソッドはすべてローカルのクライアントにアクセスできます。

ビジネスインターフェースを実装するセッション Bean は、"non-interface" ビューを表示するために記述可能です。

バグを報告する

7.2. エンタープライズ BEAN プロジェクトの作成

7.2.1. JBoss Developer Studio を使用した EJB アーカイブプロジェクトの作成

このタスクでは、JBoss Developer Studio に Enterprise JavaBeans (EJB) プロジェクトを作成する方法を説明します。

要件

- JBoss Enterprise Application Platform 6 のサーバーとサーバーランタイムが設定されている必要があります。

手順7.1 JBoss Developer Studio での EJB プロジェクトの作成

1. **新規プロジェクトの作成**
   新規 EJB プロジェクト ウィザードを開くために、**ファイル (File)** メニューで **新規 (New)** を選択し、次に **EJB プロジェクト (EJB Project)** を選択します。

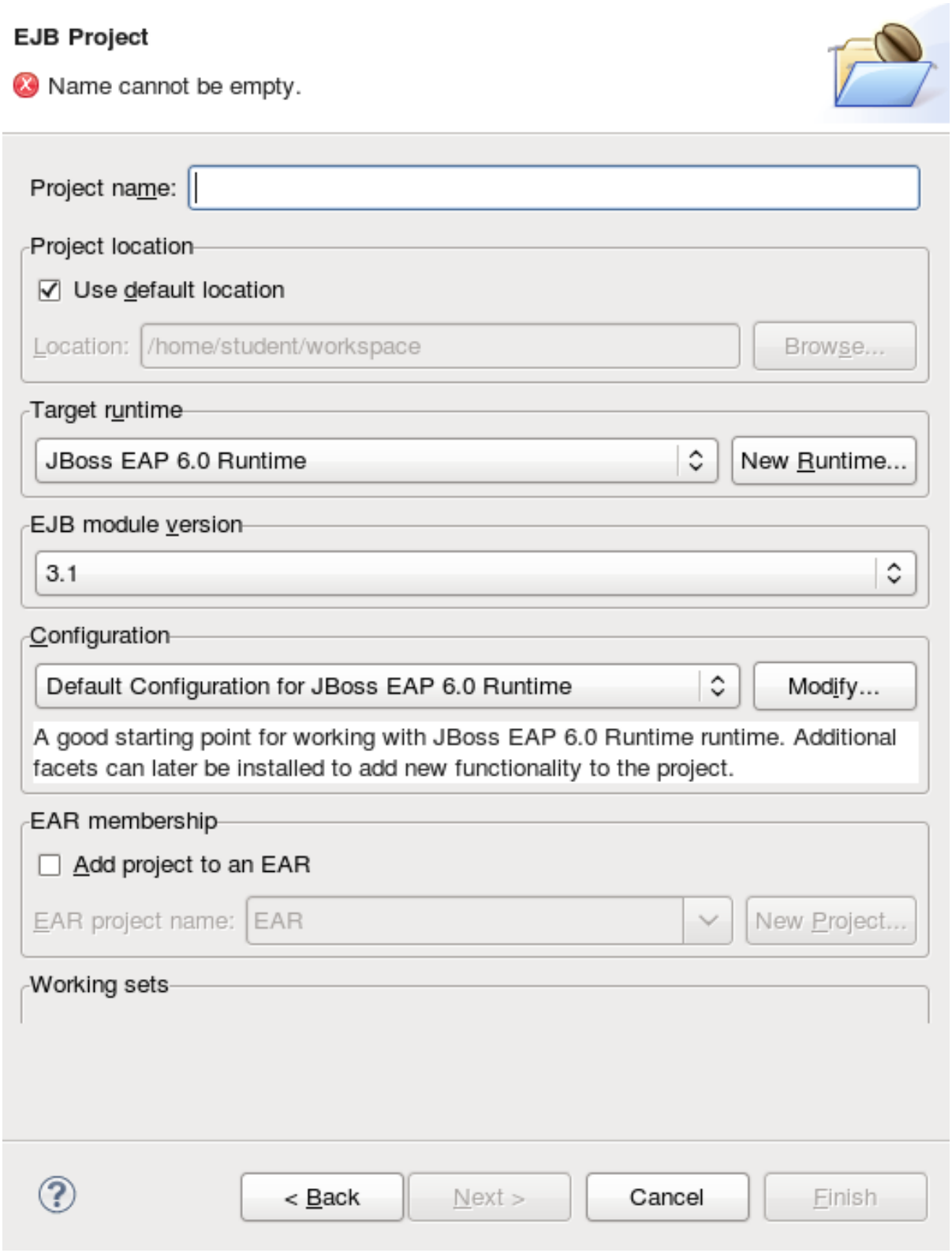


図7.1 New EJB Project ウィザード

2. **詳細の指定**
   次の詳細を入力します。

   - プロジェクト名

     JBoss Developer Studio で表示されるプロジェクト名ですが、デプロイされた JAR ファイルのデフォルトのファイル名にもなります。

   - プロジェクトの場所

プロジェクトのファイルが保存されるディレクトリです。現在のワークスペースのディレクトリがデフォルトになります。

- ターゲットランタイム

  プロジェクトに使用されるサーバーランタイムです。デプロイするサーバーによって使用される JBoss Enterprise Application Platform 6 のランタイムと同様に設定される必要があります。

- EJB モジュールバージョン。エンタープライズ Bean が準拠する EJB 仕様のバージョンになります。Red Hat は **3.1** の使用を推奨します。

- これでプロジェクトのサポート対象機能を調整できるようになります。選択したランタイムにデフォルト設定を使用します。

[Next] をクリックして作業を継続します。

3. **Java 構築設定**
   この画面では、Java ソースファイルが格納されるディレクトリや構築された出力が置かれるディレクトリをカスタマイズすることが可能です。

   この設定は変更せずに **[Next]** をクリックします。

4. **EJB モジュール設定**
   デプロイメント記述子が必要な場合は **[Generate ejb-jar.xml deployment descriptor]** チェックボックスにチェックマークを付けます。EJB 3.1 ではデプロイメント記述子は任意で、必要な場合は後で追加することが可能です。

   [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。

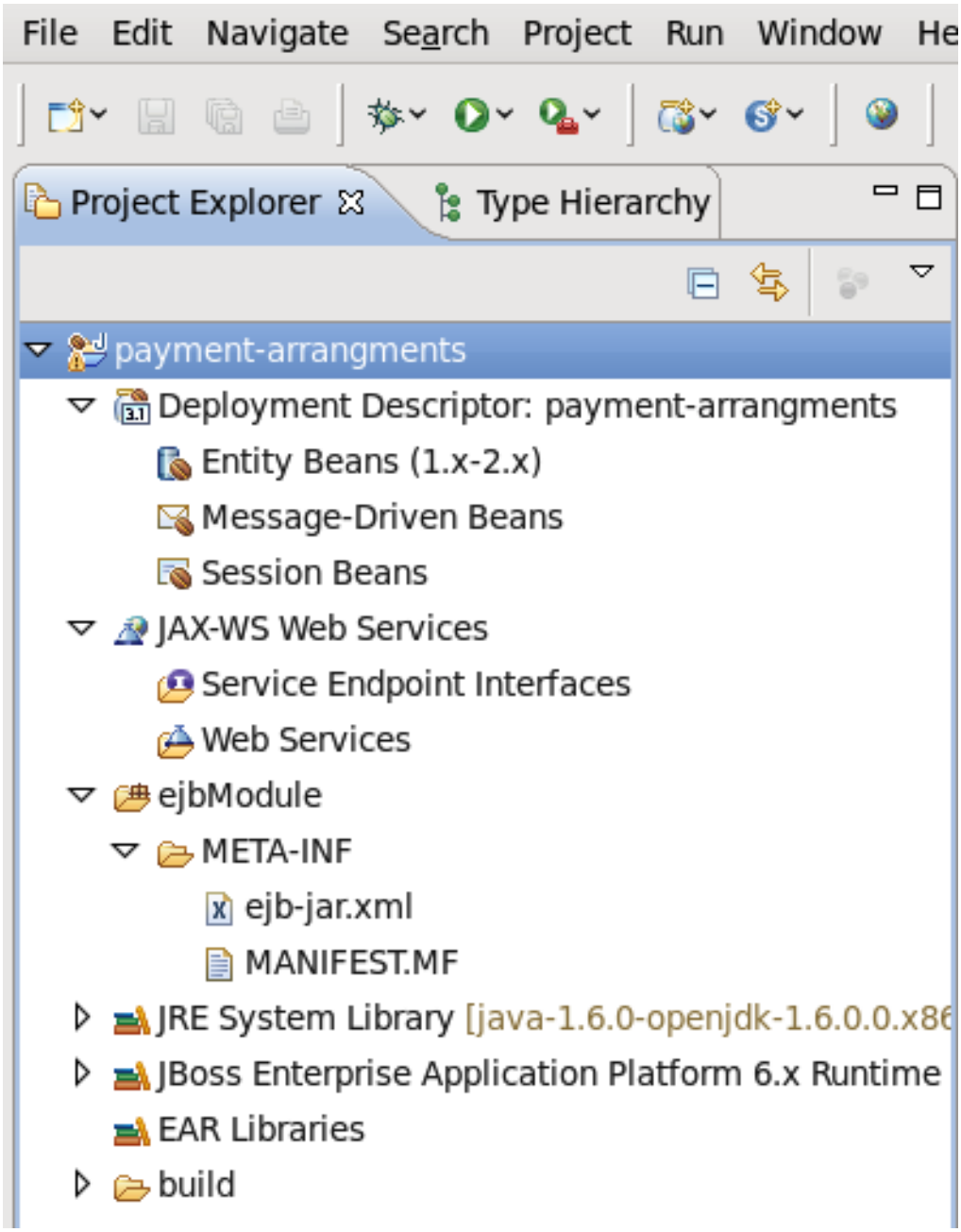


図7.2 Project Explorer の新規作成された EJB プロジェクト

5. **デプロイメントに対して構築アーティファクトをサーバーに追加する**
   サーバータブにて、構築アーティファクトをデプロイしたいサーバーを右クリックし、**[Add and Remove]** ダイアログを開きます。[Add and Remove] を選択します。

   [Available] カラムよりデプロイするリソースを選択し、**[Add]** ボタンをクリックします。リソースが **[Configured]** カラムに移動します。**[Finish]** をクリックしてダイアログを閉じます。

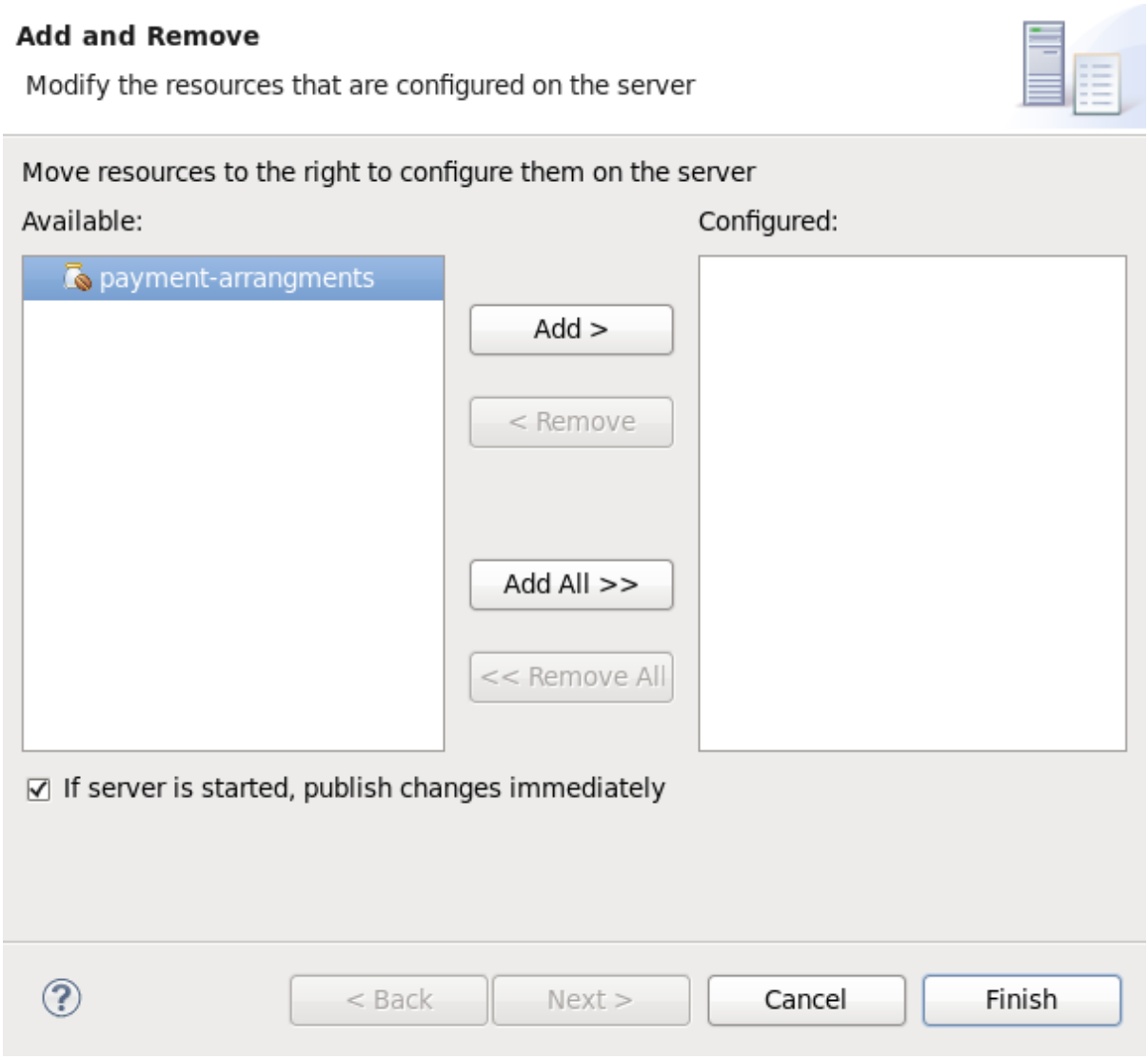


図7.3 ダイアログの追加と削除

結果

ビルドし、指定のサーバーにデプロイできる EJB プロジェクトが JBoss Developer Studio で作成されます。

プロジェクトにエンタープライズ Bean が追加されないと、JBoss Developer Studio は「An EJB module must contain one or more enterprise beans」という警告を表示します。プロジェクトにエンタープライズ Bean が 1 つ以上追加されるとこの警告は表示されないようになります。

バグを報告する

7.2.2. Maven における EJB アーカイブプロジェクトの作成

Maven を使用して JAR ファイルにパッケージ化された 1 つ以上のエンタープライズ Bean が含まれるプロジェクトを作成する方法を説明します。

前提条件

- Maven が既にインストールされている必要があります。
- Maven の基本的な使用法を理解している必要があります。

手順7.2 Maven における EJB アーカイブプロジェクトの作成

1. **Maven プロジェクトの作成**
   Maven のアーキタイプシステムと **ejb-javaee6** アーキタイプを使用して EJB プロジェクトを作成することができます。作成するには、以下のパラメーターを用いて **mvn** コマンドを実行します。

   ```
    mvn archetype:generate -
   DarchetypeGroupId=org.codehaus.mojo.archetypes -
   DarchetypeArtifactId=ejb-javaee6
   ```

   プロジェクトの **groupId**、**artifactId**、 **version**、**package** を指定するよう要求されます。

   ```
   [localhost]$ mvn archetype:generate -
   DarchetypeGroupId=org.codehaus.mojo.archetypes -
   DarchetypeArtifactId=ejb-javaee6
   [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-javaee6:1.5]
   found in catalog remote
   Define value for property 'groupId': : com.shinysparkly
   Define value for property 'artifactId': : payment-arrangments
   Define value for property 'version':  1.0-SNAPSHOT: :
   Define value for property 'package':  com.shinysparkly: :
   Confirm properties configuration:
   groupId: com.company
   artifactId: payment-arrangments
   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. **エンタープライズ Bean の追加**
   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にある **src/main/java** ディレクトリ下のプロジェクトに追加します。

3. **プロジェクトの構築**
   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で **mvn package** コマンドを実行します。このコマンドを実行すると、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。ビルドされた JAR ファイルには ***artifactId-version*.jar** という名前が付けられ、**target/** ディレクトリに置かれます。

結果: 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す。アプリケーションサーバーへデプロイすることができるエンタープライズ Bean と JAR ファイルがこのプロジェクトに含まれるようにすることが可能です。

バグを報告する

7.2.3. EJB プロジェクトが含まれる EAR プロジェクトの作成

EJB プロジェクトが含まれる新しいエンタープライズアーカイブ (EAR) プロジェクトを JBoss Developer Studio で作成する方法を説明します。

要件

- JBoss Enterprise Application Platform 6 のサーバーとサーバーランタイムが設定されている必要があります。詳細は「JBoss Enterprise Application Platform 6 サーバーの JBoss Developer Studio への追加」を参照してください。

手順7.3 EJB プロジェクトが含まれる EAR プロジェクトの作成

1. **新しい EAR アプリケーションプロジェクトウィザードを開く**
   [File] メニューより **[New]**、**[Project]** の順に選択すると、**[New Project]** ウィザードが表示されます。**[Java EE/Enterprise Application Project]** を選択し、**[Next]** をクリックします。

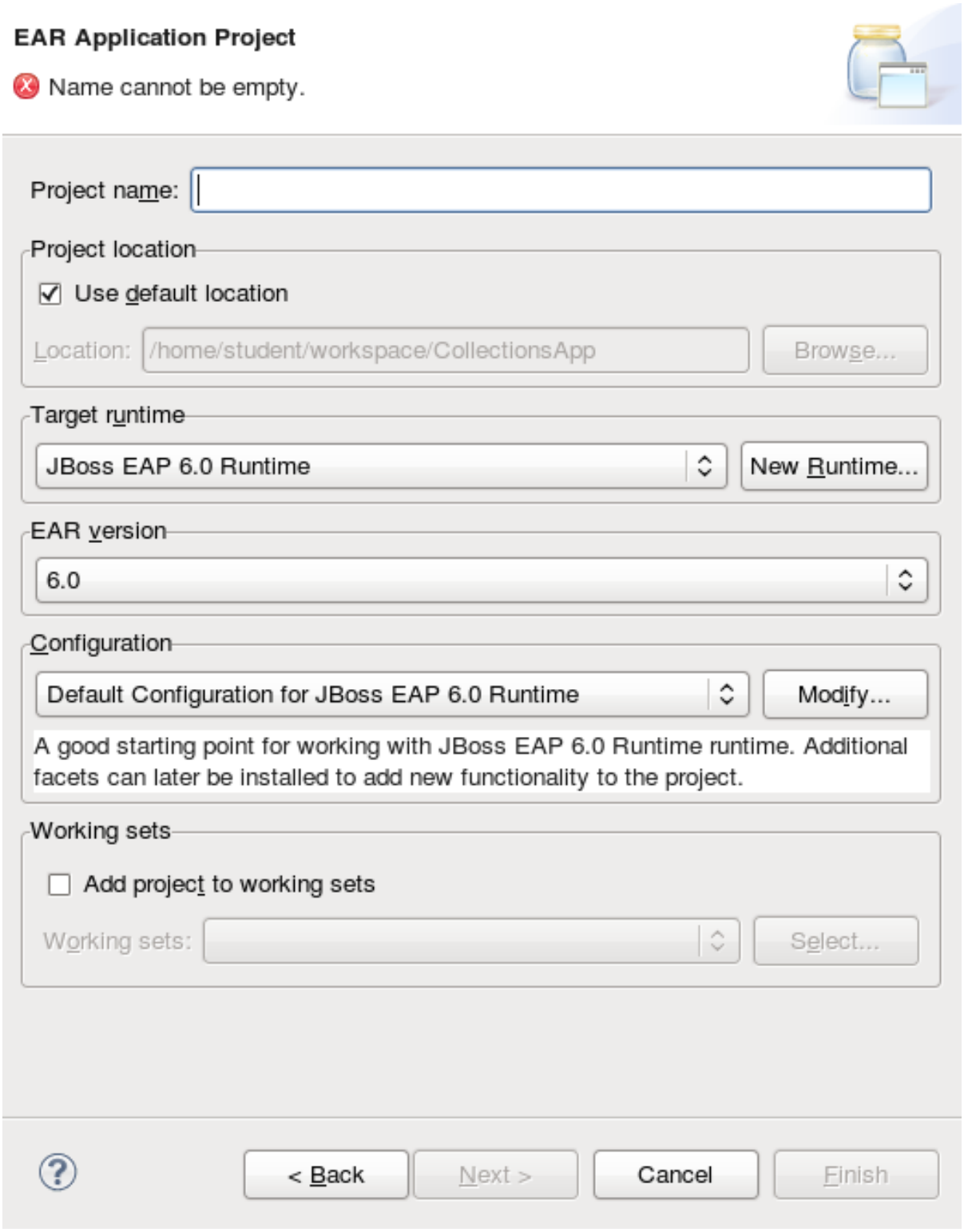


図7.4 新しい EAR アプリケーションウィザード

2. **詳細の入力**
   次の詳細を入力します。

   - プロジェクト名

     JBoss Developer Studio で表示されるプロジェクト名ですが、デプロイされた EAR ファイルのデフォルトのファイル名にもなります。

   - プロジェクトの場所

プロジェクトのファイルが保存されるディレクトリです。現在のワークスペースのディレクトリがデフォルトになります。

- ターゲットランタイム

  プロジェクトに使用されるサーバーランタイムです。デプロイするサーバーによって使用される JBoss Enterprise Application Platform 6 のランタイムと同様に設定される必要があります。

- EAR バージョン

  プロジェクトが準拠する Java Enterprise Edition 仕様のバージョンになります。Red Hat は **6** の使用を推奨します。

- これでプロジェクトのサポート対象機能を調整できるようになります。選択したランタイムにデフォルト設定を使用します。

[Next] クリックして作業を継続します。

3. **新しい EJB モジュールの追加**
   新しいモジュールはウィザードの **Enterprise Application** ページより追加することができます。次の手順に従って新しい EJB プロジェクトをモジュールとして追加します。

   a. **新しい EJB モジュールの追加**
   [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ます。**[Enterprise Java Bean]** を選択し、**[Next]** をクリックすると **[New EJB Project]** ウィザードが表示されます。

   b. **EJB プロジェクトの作成**
   New EJB Project ウィザードは、新しいスタンドアローン EJB プロジェクトを作成するために使用するウィザードと同じで、「JBoss Developer Studio を使用した EJB アーカイブプロジェクトの作成」 に説明されています。

   プロジェクト作成のために最低限必要な情報は次の通りです。

   - プロジェクト名
   - ターゲットランタイム
   - EJB モジュールのバージョン
   - 設定

   ウィザードの他の手順はすべて任意の手順となります。**[Finish]** をクリックして EJB プロジェクトの作成を完了します。

   新規作成された EJB プロジェクトは Java EE モジュールの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。

4. **任意の作業: application.xml デプロイメント記述子の追加**
   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付けます。

5. **Finish のクリック**
   EJB プロジェクトと EAR プロジェクトの 2 つの新しいプロジェクトが表示されます。

6. **デプロイメントに対して構築アーティファクトをサーバーに追加する**

[Servers] タブにて、構築アーティファクトをデプロイしたいサーバーを右クリックし、**[Add and Remove]** ダイアログを開きます。**[Add and Remove]** を選択します。

[Available] カラムよりデプロイする EAR リソースを選択し、 **[Add]** ボタンをクリックします。リソースが **[Configured]** カラムに移動します。**[Finish]** をクリックしてダイアログを閉じます。

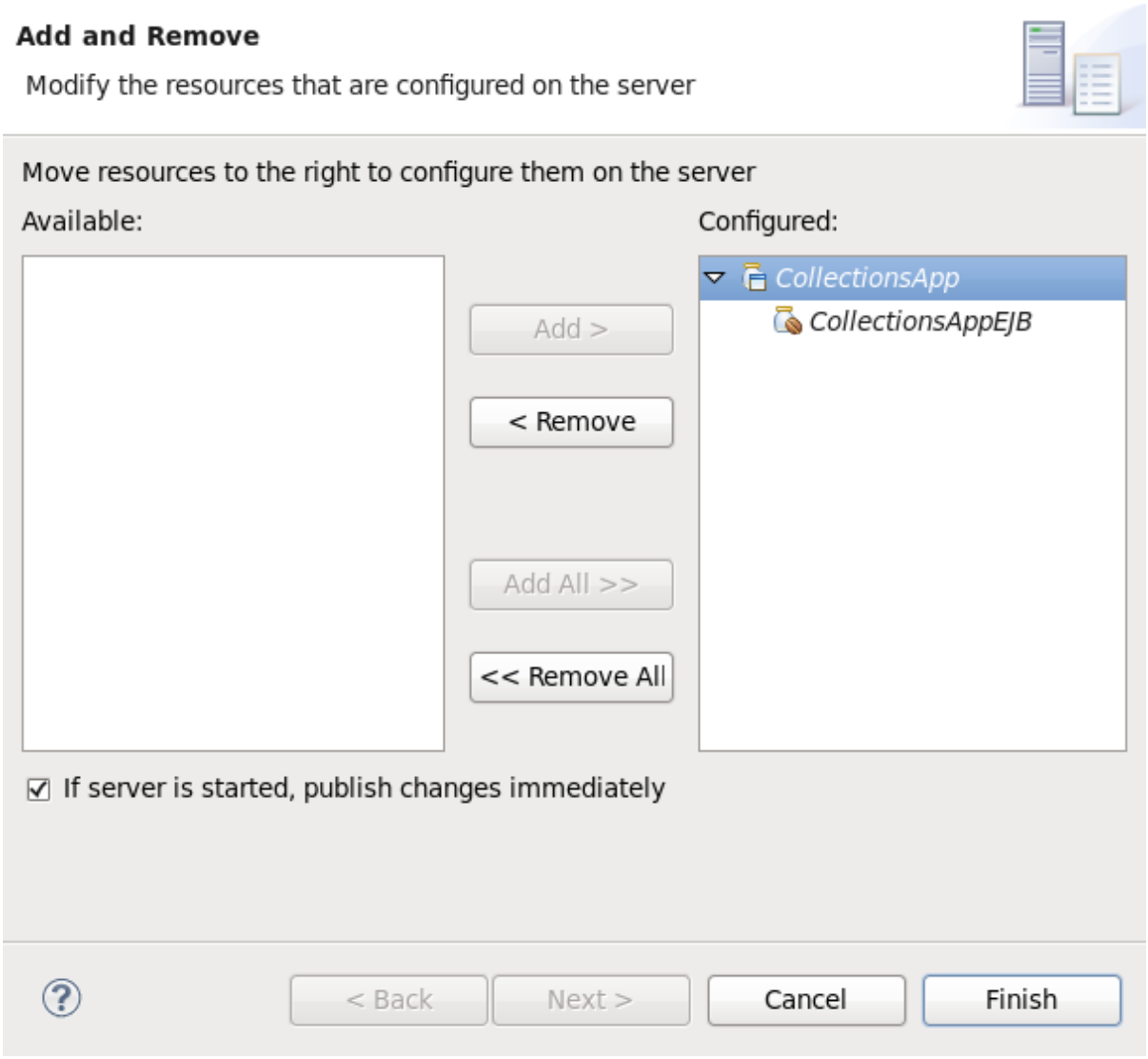


図7.5 ダイアログの追加と削除

結果

メンバーの EJB プロジェクトを持つ Enterprise Application プロジェクトが作成されます。この Enterprise Application プロジェクトはビルドされ、EJB サブデプロイメントが含まれる単一の EAR デプロイメントとして指定のサーバーにデプロイされます。

バグを報告する

7.2.4. 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がない状態で作成された EJB プロジェクトに EJB デプロイメント記述子を追加することができます。次の手順に従って追加します。

前提条件

- EJB デプロイメント記述子を追加したい EJB プロジェクトが JBoss Developer Studio に存在している必要があります。

手順7.4 EJB プロジェクトにデプロイメント記述子を追加する

1. **プロジェクトを開く**
   JBoss Developer Studio でプロジェクトを開きます。

2. **デプロイメント記述子の追加**
   プロジェクトビューの Deployment Descriptor フォルダーを右クリックし、**[Generate Deployment Descriptor Stub]** を選択します。

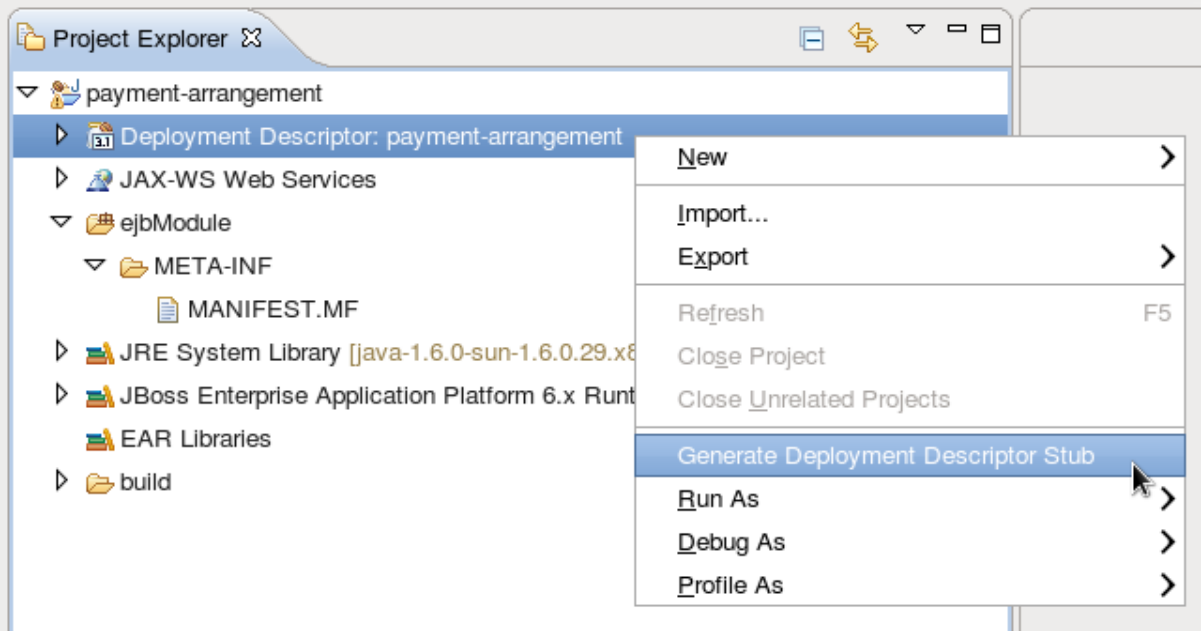


図7.6 デプロイメント記述子の追加

新しいファイル **`ejb-jar.xml`** が **`ejbModule/META-INF/`** に作成されます。

バグを報告する

7.3. セッション BEAN

7.3.1. セッション Bean

セッション Bean は、関連の業務プロセスやタスクのセットをカプセル化し、要求した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 には、ステートレス、ステートフル、シングルトンの 3 種類があります。

バグを報告する

7.3.2. ステートレスセッション Bean

ステートレスセッション Bean は最もシンプルですが、幅広く利用されているセッション Bean です。クライアントアプリケーションへの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ステートは保持しません。各メソッドは、セッション Bean 内で共有ステートに依存しない完全タスクです。ステートがないため、アプリケーションサーバーは各メソッド呼び出しが同じインスタンスで実行されているか確認する必要がありません。結果、ステートレス Bean の効率と拡張性は非常に高くなります。

バグを報告する

7.3.3. ステートフルセッション Bean

ステートフルセッション Bean はエンタープライズ Bean でビジネスメソッドをクライアントアプリケーションに渡し、クライアントとの会話の状態を維持します。複数のステップ (メソッド呼び出し) を踏んで実行する必要のあるタスクにこれらの Bean を利用してください。それぞれのステップでは、1 つ前のステップの状態を維持します。アプリケーションサーバーは、各クライアントがメソッド呼び出しごとに同じステートフルセッション Bean のインスタンスを受け取るようにします。

バグを報告する

7.3.4. 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 1 回インスタンス化されるセッション Bean で、1 つのシングルトン Bean に対するクライアント要求はすべて同じインスタンスへ送信されます。シングルトン Bean はシングルトンデザインパターンの実装であり、Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides によって執筆され、1994 年に Addison-Wesley より出版された 『Design Patterns: Elements of Reusable Object-Oriented Software』 で説明されています。

シングルトン Bean はセッション Bean の型で最も小さいメモリーフットプリントを提供しますが、スレッドセーフである必要があります。EJB 3.1 はコンテナ管理の並行性 (Container-Managed Concurrency、CMC) を提供し、開発者がスレッドセーフのシングルトン Bean を簡単に実装できるようにします。CMC の柔軟性が足りない場合は、従来のマルチスレッドコード (Bean 管理の並行性、BMC) を使用してシングルトン Bean を書くことも可能です。

バグを報告する

7.3.5. JBoss Developer Studio のプロジェクトにセッション Bean を追加する

JBoss Developer Studio にはエンタープライズ Bean クラスを即座に作成できる複数のウィザードがあります。以下は、JBoss Developer Studio のウィザードを使用してプロジェクトにセッション Bean を追加する手順になります。

前提条件

- 1 つ以上のセッション Bean を追加したい EJB または動的 Web プロジェクトが JBoss Developer Studio に存在する必要があります。

手順7.5 JBoss Developer Studio のプロジェクトにセッション Bean を追加する

1. **プロジェクトを開く**
   JBoss Developer Studio でプロジェクトを開きます。

2. **[Create EJB 3.x Session Bean] ウィザードを開く**
   [Create EJB 3.x Session Bean] ウィザードを開くために、**[File]** メニューへ移動し、**[New]** を選択してから **[Session Bean (EJB 3.x)]** を選択します。

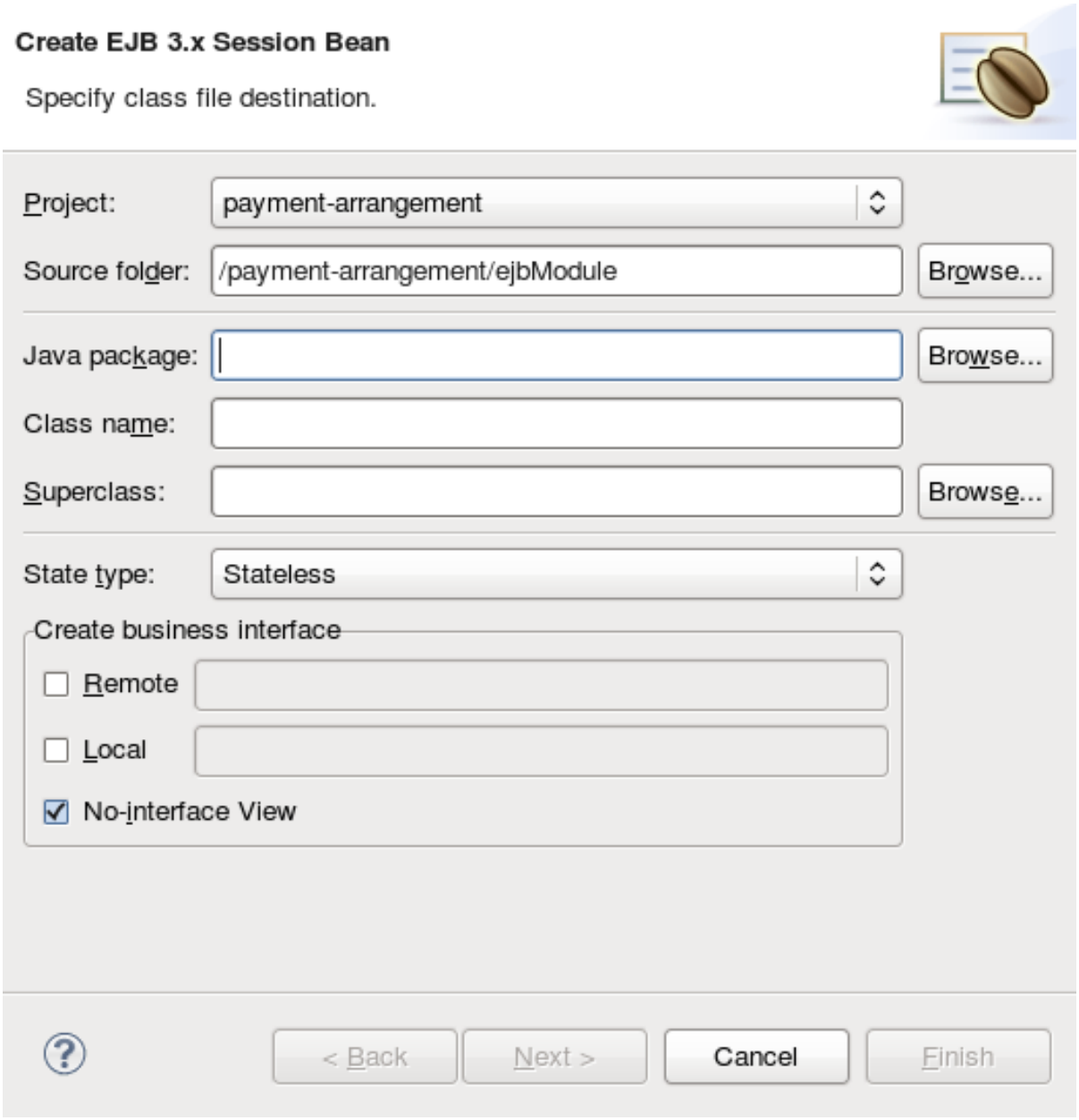


図7.7 [Create EJB 3.x Session Bean] ウィザード

3. **クラス情報の指定**
   次の詳細を入力します。

   - プロジェクト

     正しいプロジェクトが選択されているか検証します。

   - ソースホルダー

     Java ソースファイルが作成されるフォルダーになります。通常、変更する必要はありません。

   - パッケージ

     クラスが属するパッケージを指定します。

   - クラス名

     セッション Bean になるクラスの名前を指定します。

- スーパークラス

  セッション Bean クラスはスーパークラスより継承することができます。セッションにスーパークラスがあるかどうかをここに指定します。

- ステートタイプ

  セッション Bean のステートタイプ (ステートレス、ステートフル、シングルトン) を指定します。

- ビジネスインターフェース

  デフォルトでは No-interface ボックスにチェックマークが付けられているため、インターフェースは作成されません。定義したいインターフェースのボックスにチェックマークを付け、必要な場合は名前を調整します。

  Web アーカイブ (WAR) のエンタープライズ Bean は EJB 3.1 Lite のみをサポートするため、リモートビジネスインターフェースは含まれません。

[Next] をクリックします。

4. **セッション Bean の特定情報**
   ここに追加情報を入力してセッション Bean を更にカスタマイズすることが可能です。ここで情報を変更する必要はありません。

   変更できる項目は次の通りです。

   - Bean 名。
   - マッピングされた名前。
   - トランザクションタイプ (コンテナ管理または Bean 管理)。
   - Bean が実装しなければならない追加のインターフェースを入力できます。
   - 必要な場合は、EJB 2.x のホームインターフェースやコンポーネントインターフェースを指定することもできます。

5. **完了**
   [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。指定された場合、新しいビジネスインターフェースのファイルも作成されます。

結果: 新しいセッション Bean がプロジェクトに追加されます。

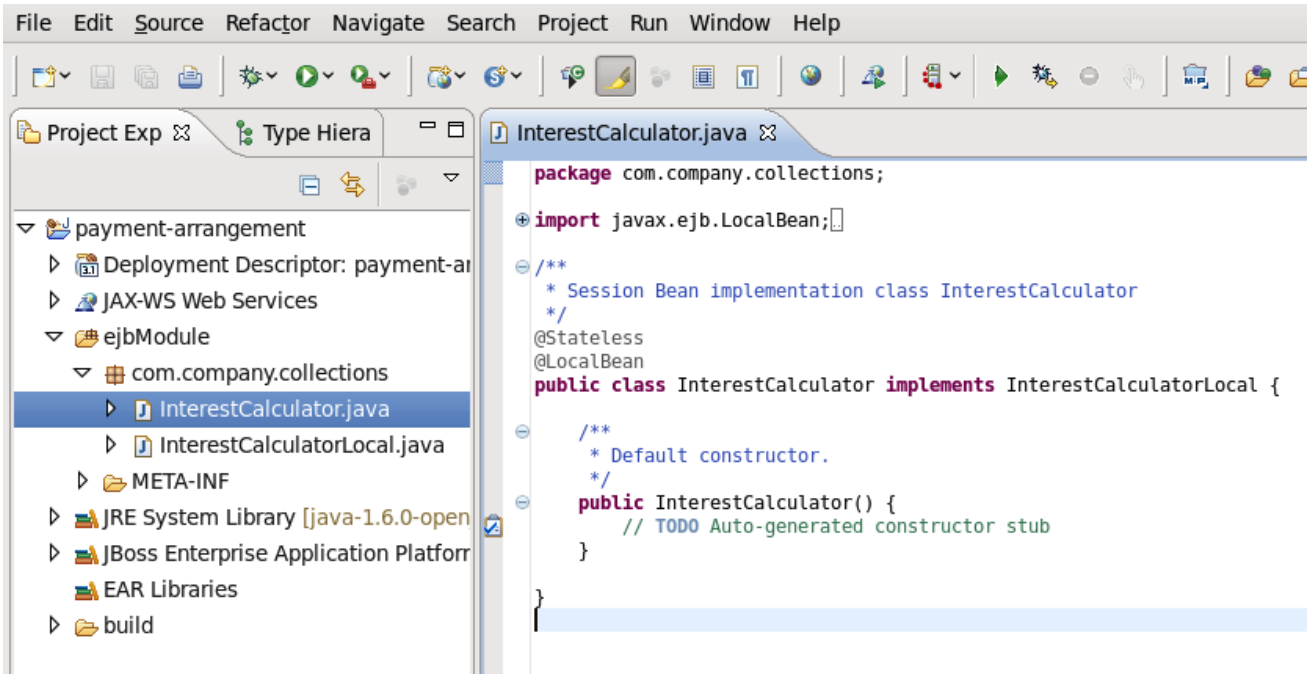


図7.8 JBoss Developer Studio の新しいセッション Bean

バグを報告する

7.4. メッセージ駆動型 BEAN

7.4.1. メッセージ駆動型 Bean

メッセージ駆動型 Bean (MDB) は、アプリケーション開発に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挿入されるか、クライアントコードから呼び出されますが、Java Messaging Service (JMS) サーバーなどのメッセージングサービスからメッセージを受け取ることによってトリガーされます。Java EE 6 仕様では JMS がサポートされている必要がありますが、他のメッセージングシステムをサポートすることもできます。

バグを報告する

7.4.2. リソースアダプター

リソースアダプターは、 Java Connector Architecture (JCA) 仕様を使用して Java EE アプリケーションとエンタープライズ情報システム (EIS) との間の通信を提供するデプロイ可能な Java EE コンポーネントです。通常、リソースアダプターは EIS のベンダーによって提供されるため、ベンダーの製品と Java EE アプリケーションとの統合は容易になります。

エンタープライズ情報システムは、組織内における他のあらゆるソフトウェアシステムのことです。例としては、エンタープライズリソースプランニング (ERP) システム、データベースシステム、電子メールサーバー、商用メッセージングシステムなどが挙げられます。

リソースアダプターは、JBoss Enterprise Application Platform 6 にデプロイできる Resource Adapter Archive (RAR) ファイルでパッケージ化されます。また、RAR ファイルは、Enterprise Archive (EAR) デプロイメントにも含まれていることがあります。

バグを報告する

7.4.3. JBoss Developer Studio に JMS ベースのメッセージ駆動型 Bean を作成する

JBoss Developer Studio のプロジェクトに JMS ベースのメッセージ駆動型 Bean を追加する手順は次の通りです。この手順では、アノテーションを使用する EJB 3.x メッセージ駆動型 Bean を作成します。

前提条件

1. JBoss Developer Studio で既存のプロジェクトが開かれていなければなりません。
2. Bean がリッスンする JMS 宛先の名前とタイプを認識している必要があります。
3. Bean がデプロイされる JBoss Enterprise Application Platform の設定で Java メッセージングサービス (JMS) のサポートが有効になっている必要があります。

手順7.6 JBoss Developer Studio に JMS ベースのメッセージ駆動型 Bean を追加する

1. **`[Create EJB 3.x Message-Driven Bean]` ウィザードを開く**
   [File] → **[New]** → **[Other]** と移動します。**`[EJB/Message-Driven Bean (EJB 3.x)]`** を選択し、 **`[Next]`** ボタンをクリックします。

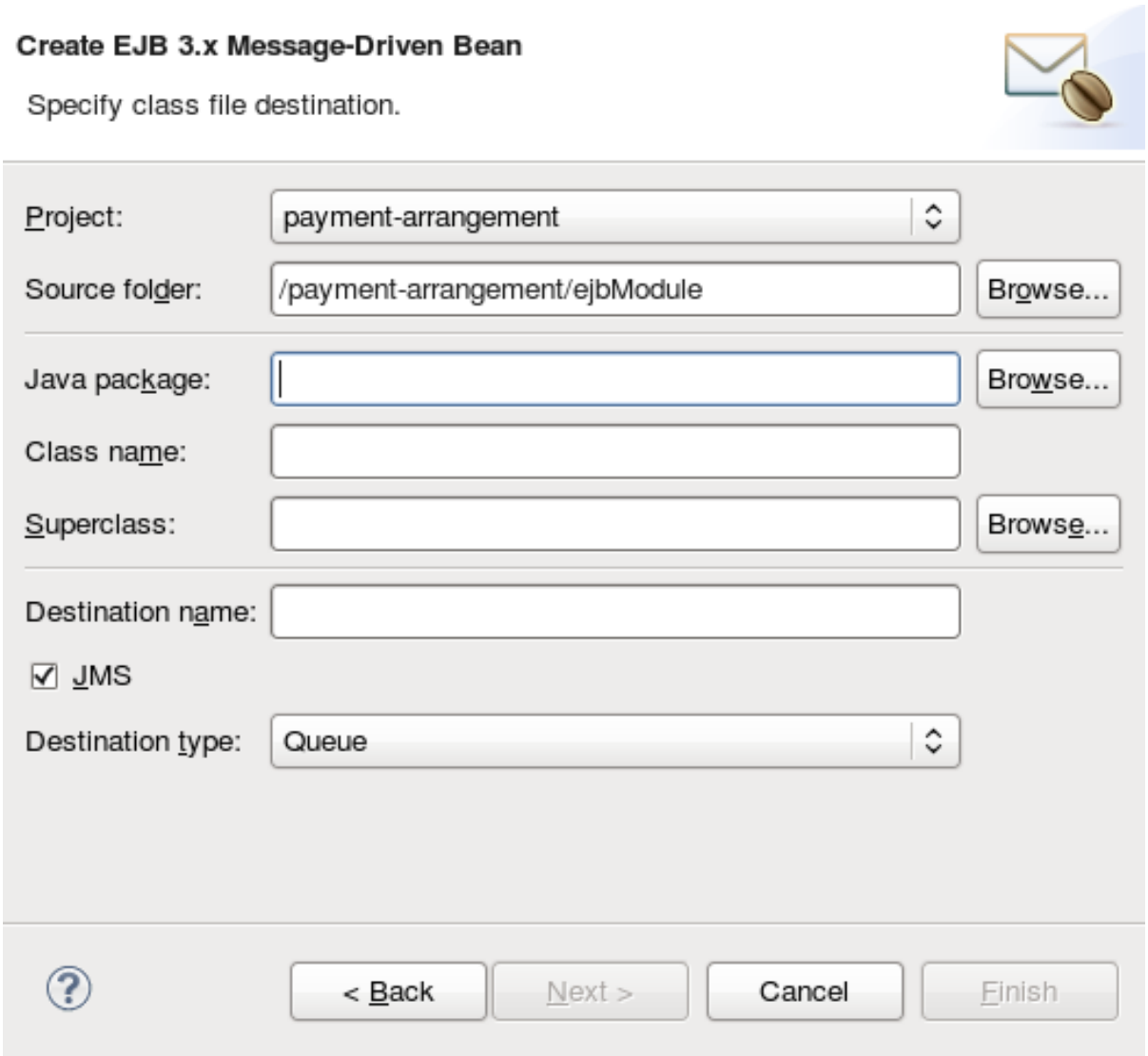


図7.9 Create EJB 3.x Message-Driven Bean ウィザード

2. **クラスファイルの宛先詳細の指定**
   Bean クラスに対して指定する詳細のセットは、プロジェクト、Java クラス、メッセージの宛先の 3 つがあります。

プロジェクト

- **[Workspace]** に複数のプロジェクトが存在する場合は、**[Project]** メニューで正しいプロジェクトが選択されるようにしてください。
- 新しい Bean のソースファイルが作成されるフォルダーは、選択されたディレクトリ下の **ejbModule** に作成されます。特定の要件がある場合のみこのフォルダーを変更します。

Java クラス

- 必須のフィールドは **[Java package]** と **[class name]** になります。
- アプリケーションのビジネスロジックがスーパークラスを必要とする場合を除き、**[Superclass]** を入力する必要はありません。

メッセージの宛先

JMS ベースのメッセージ駆動型 Bean に提供しなければならない詳細は次の通りです。

- **[Destination name]**。Bean が応答するメッセージに含まれるキューまたはトピック名です。
- 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。
- **[Destination type]** を必要に応じて **[Queue]** または **[Topic]** に設定します。

[Next] ボタンをクリックします。

3. **メッセージ駆動型 Bean に固有の情報の入力**
以下のデフォルト値はコンテナ管理トランザクションを使用する JMS ベースのメッセージ駆動型 Bean に適するデフォルト値となります。
   - Bean が Bean 管理トランザクションを使用する場合はトランザクションタイプを 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 名を変更します。
   - JMS メッセージリスナーインターフェースが表示されるはずです。インターフェースがアプリケーションのビジネスロジックに固有する場合を除き、インターフェースを追加したり削除したりする必要はありません。
   - メソッドスタブ作成のチェックボックスはそのまま選択された状態にしてきます。

   [Finish] ボタンをクリックします。

結果: デフォルトのコンストラクターのスタブメソッドと **onMessage()** メソッドによってメッセージ駆動型 Bean が作成されます。JBoss Developer Studio のエディターウィンドウが対応するファイルによって開かれます。

バグを報告する

7.5. セッション BEAN の呼び出し

7.5.1. JNDI を使用したリモートでのセッション Bean の呼び出し

このタスクは、JNDI を使用してセッション Bean の呼び出すリモートクライアントへサポートを追加する方法を説明します。Maven を使用してプロジェクトがビルドされていることが前提となります。

ejb-remote クイックスタートには、この機能のデモを行う Maven プロジェクトが含まれています。このクイックスタートには、デプロイするセッション Bean のプロジェクトとリモートクライアントのプロジェクトの両方が含まれています。下記のコード例はリモートクライアントのプロジェクトから引用されています。

このタスクでは、セッション Bean に認証の必要がないことが前提となっています。

要件

始める前に、次の前提条件を満たしている必要があります。

- Maven プロジェクトが作成され、使用できる状態です。
- JBoss Enterprise Application Platform 6 の Maven リポジトリが既に追加されています。
- 呼び出しするセッション Bean が既にデプロイされています。
- デプロイされたセッション Bean がリモートビジネスインターフェースを実装します。
- セッション Bean のリモートビジネスインターフェースは Maven 依存関係として使用できます。リモートビジネスインターフェースが JAR ファイルとしてのみ使用できる場合は、JAR をアーティファクトとして Maven リポジトリに追加することが推奨されます。手順については、http://maven.apache.org/plugins/maven-install-plugin/usage.html にある Maven ドキュメントの **install:install-file** ゴールを参照してください。
- セッション Bean をホストするサーバーのホスト名と JNDI ポートを覚えておく必要があります。

リモートクライアントよりセッション Bean を呼び出すには、最初にプロジェクトを適切に設定する必要があります。

手順7.7 セッション Bean のリモート呼び出しに対する Maven プロジェクト設定を追加する

1. **必要なプロジェクト依存関係の追加**
   必要な依存関係が含まれるようにするため、プロジェクトの **pom.xml** を更新する必要があります。

2. **jboss-ejb-client.properties ファイルの追加**
   JBoss EJB クライアント API は、JNDI サービスの接続情報が含まれる **jboss-ejb-client.properties** という名前のプロジェクトのルートにファイルがあることを想定します。このファイルを以下の内容と共にプロジェクトの **src/main/resources/** ディレクトリーに追加します。

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLE
D=false

remote.connections=default

remote.connection.default.host=localhost
remote.connection.default.port = 4447
```

```
remote.connection.default.connect.options.org.xnio.Options.SASL_POLI
CY_NOANONYMOUS=false
```

ホスト名とポートを変更してサーバーと一致するようにします。**4447** がデフォルトのポート番号です。 安全な接続の場合、**SSL_ENABLED** 行を **true** に設定し、 **SSL_STARTTLS** 行をアンコメントします。コンテナ内のリモーティングインターフェースは同じポートを使用して安全な接続と安全でない接続をサポートします。

3. **リモートビジネスインターフェースの依存関係の追加**
   セッション Bean のリモートビジネスインターフェースに対する **pom.xml** に Maven の依存関係を追加します。

```
<dependency>
   <groupId>org.jboss.as.quickstarts</groupId>
   <artifactId>jboss-as-ejb-remote-server-side</artifactId>
   <type>ejb-client</type>
   <version>${project.version}</version>
</dependency>
```

これでプロジェクトが適切に設定されたため、コードを追加してセッション Bean へアクセスしたり呼び出しすることができるようになりました。

手順7.8 JNDI を使用して Bean プロキシを取得し、Bean のメソッドを呼び出す

1. **チェック例外の処理**
   次のコードに使用されるメソッドの 2 つ (**InitialContext()** および **lookup()**) は、タイプ **javax.naming.NamingException** のチェック済み例外を持っています。これらのメソッド呼び出しは **NamingException** をキャッチする try/catch ブロックか、**NamingException** のスローが宣言されたメソッドに存在する必要があります。**ejb-remote** クイックスタートでは、2 番目の方法を使用します。

2. **JNDI コンテキストの作成**
   JNDI コンテキストオブジェクトはサーバーよりリソースを要求するメカニズムを提供します。次のコードを使用して JNDI コンテキストを作成します。

```
final Hashtable jndiProperties = new Hashtable();
jndiProperties.put(Context.URL_PKG_PREFIXES,
"org.jboss.ejb.client.naming");
final Context context = new InitialContext(jndiProperties);
```

   JNDI サービスの接続プロパティーは **jboss-ejb-client.properties** ファイルより読み取られます。

3. **JNDI コンテキストの lookup() メソッドを使用した Bean プロキシの取得**
   Bean プロキシの **lookup()** メソッドを呼び出し、必要なセッション Bean の JNDI 名 へ渡します。これにより、呼び出したいメソッドが含まれるリモートビジネスインターフェースのタイプへキャストされなければならないオブジェクトが返されます。

```
final RemoteCalculator statelessRemoteCalculator =
(RemoteCalculator) context.lookup(
```

```
        "ejb:/jboss-as-ejb-remote-server-side/CalculatorBean!" +
        RemoteCalculator.class.getName());
```

セッション Bean の JNDI 名は特別な構文によって定義されます。

4. **呼び出しメソッド**
   プロキシ Bean オブジェクトを取得したため、リモートビジネスインターフェースに含まれるすべてのメソッドを呼び出しすることができます。

```
int a = 204;
int b = 340;
System.out.println("Adding " + a + " and " + b + " via the remote
stateless calculator deployed on the server");
int sum = statelessRemoteCalculator.add(a, b);
System.out.println("Remote calculator returned sum = " + sum);
```

   メソッド呼び出し要求が実行されるサーバー上で、プロキシ Bean がメソッド呼び出し要求をセッション Bean へ渡します。結果はプロキシ Bean へ返され、プロキシ Bean によって結果が呼び出し側へ返されます。プロキシ Bean とリモートセッション Bean 間の通信は呼び出し側に透過的です。

これで、Maven プロジェクトを設定してリモートサーバー上で呼び出しを行うセッション Bean をサポートし、JNDI を使用してサーバーより読み出したプロキシ Bean を使用してセッション Bean メソッドを呼び出すコードを作成できるようになりました。

バグを報告する

7.5.2. EJB クライアントコンテキストについて

JBoss Enterprise Application Platform 6 には、リモート EJB 呼び出しを管理する 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、1 つまたは複数のスレッドで同時に関連付けたり、使用したりできる EJBClientContext を使用します。つまり、EJBClientContext には任意の数の EJB レシーバーを含めることができます。EJB レシーバーは、EJB 呼び出しを処理できるサーバーとの通信方法を知っているコンポーネントです。一般的に、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。

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

EJB クライアント API の観点から、リモートクライアントのタイプに応じて、1 つの JVM 内には複数の EJBClientContext が存在することがあります。

スタンドアロンアプリケーションは通常、任意の数の EJB レシーバーにより支援されることがある単一の EJBClientContext を持ちますが、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が複数の EJBClientContext を持つ場合、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 Enterprise Application Platform 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、デプロイされた各アプリケーションは、対応する EJB クライアントコンテキストを持ちます。このアプリケーションが別の EJB を呼び出すと、適切な EJB レシーバーを見つけるために、対応する EJB クライアントコンテキストが使用され、呼び出しが処理されます。

バグを報告する

7.5.3. 単一 EJB コンテキストを使用する場合の留意事項

概要

スタンドアロンリモートクライアントで単一 EJB クライアントコンテキストを使用する場合は、アプリケーション要件を考慮する必要があります。異なるタイプのリモートクライアントの詳細については、「EJB クライアントコンテキストについて」を参照してください。

単一 EJB クライアントコンテキストを持つリモートスタンドアロンの一般的なプロセス

一般的に、リモートスタンドアロンクライアントは任意の数の EJB レシーバーにより支援された唯一の EJB クライアントコンテキストを持っています。以下に、スタンドアロンリモートクライアントアプリケーションの例を示します。

```
public class MyApplication {
    public static void main(String args[]) {
        final javax.naming.Context ctxOne = new
javax.naming.InitialContext();
        final MyBeanInterface beanOne =
ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...
    }
}
```

リモートクライアント JNDI ルックアップは、通常 **jboss-ejb-client.properties** ファイルにより支援され、このファイルは EJB クライアントコンテキストと EJB レシーバーをセットアップするために使用されます。また、この設定には、セキュリティークレデンシャルが含まれ、このセキュリティークレデンシャルは JBoss Enterprise Application Platform サーバーに接続する EJB レシーバーを作成するために使用されます。上記のコードが呼び出された場合、EJB クライアント API は EJB クライアントコンテキストを探します。この EJB クライアントコンテキストは EJB 呼び出し要求を受け取り処理する EJB レシーバーを選択するために使用されます。この場合は、EJB クライアントコンテキストが 1 つしかないため、Bean を呼び出すためにそのコンテキストが上記のコードにより使用されます。JNDI をリモートで使用してセッション Bean を呼び出す手順の詳細については、「JNDI を使用したリモートでのセッション Bean の呼び出し」を参照してください。

リモートスタンドアロンクライアントは異なるクレデンシャルを必要とする

ユーザーアプリケーションが Bean を複数回呼び出す場合に、異なるセキュリティークレデンシャルを使用して JBoss Enterprise Application Platform サーバーに接続したいことがあります。以下に、同じ Bean を 2 回呼び出すスタンドアロンリモートクライアントアプリケーションの例を示します。

```
public class MyApplication {
    public static void main(String args[]) {
        // Use the "foo" security credential connect to the server and
invoke this bean instance
        final javax.naming.Context ctxOne = new
javax.naming.InitialContext();
        final MyBeanInterface beanOne =
ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...

        // Use the "bar" security credential to connect to the server and
invoke this bean instance
        final javax.naming.Context ctxTwo = new
javax.naming.InitialContext();
```

```
        final MyBeanInterface beanTwo =
ctxTwo.lookup("ejb:app/module/distinct/bean!interface");
        beanTwo.doSomething();
        ...
    }
}
```

この場合、アプリケーションは同じサーバーインスタンスに接続してそのサーバーにホストされた EJB を呼び出し、サーバーに接続する際に 2 つの異なるクレデンシャルを使用します。クライアントアプリケーションは単一の EJB クライアントコンテキストを持ち、各サーバーインスタンスに対して EJB レシーバーを 1 つしか持つことができないため、上記のコードはクレデンシャルを 1 つだけ使用してサーバーに接続し、コードはアプリケーションの期待どおりに実行されません。

解決法

スコープ EJB クライアントコンテキストを使用するとこの問題を解決できます。スコープ EJB クライアントコンテキストにより、EJB クライアントコンテキストと、関連する JNDI コンテキスト (一般的に EJB 呼び出しに使用されます) を細かく制御できるようになります。スコープ EJB クライアントコンテキストの詳細については、「スコープ EJB クライアントコンテキストの使用」と「スコープ EJB クライアントコンテキストを使用した EJB の設定」 を参照してください。

バグを報告する

7.5.4. スコープ EJB クライアントコンテキストの使用

概要

JBoss Enterprise Application Platform 6 の初期バージョンで EJB を呼び出すには、通常 JNDI コンテキストを作成し、PROVIDER_URL (ターゲットサーバーを示します) に渡します。JNDI コンテキストを使用してルックアップされた EJB プロキシに対して行われたすべての呼び出しは最終的にそのサーバーに対して行われます。スコープ EJB クライアントコンテキストにより、ユーザーアプリケーションは特定の呼び出しに使用される EJB レシーバーを制御できます。

リモートスタンドアロンクライアントでスコープ EJB クライアントコンテキストを使用する

スコープ EJB クライアントコンテキストが導入される前に、コンテキストは通常クライアントアプリケーションにスコープ指定されていました。スコープクライアントコンテキストにより、EJB クライアントコンテキストを JNDI コンテキストでスコープ指定できるようになりました。以下に、スコープ EJB クライアントコンテキストを使用して同じ Bean を 2 回呼び出すスタンドアロンリモートクライアントアプリケーションの例を示します。

```
public class MyApplication {
    public static void main(String args[]) {

        // Use the "foo" security credential connect to the server and
invoke this bean instance
        final Properties ejbClientContextPropsOne =
getPropsForEJBClientContextOne():
        final javax.naming.Context ctxOne = new
javax.naming.InitialContext(ejbClientContextPropsOne);
        final MyBeanInterface beanOne =
ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...

        // Use the "bar" security credential to connect to the server and
invoke this bean instance
```

```
        final Properties ejbClientContextPropsTwo =
getPropsForEJBClientContextTwo():
        final javax.naming.Context ctxTwo = new
javax.naming.InitialContext(ejbClientContextPropsTwo);
        final MyBeanInterface beanTwo =
ctxTwo.lookup("ejb:app/module/distinct/bean!interface");
        beanTwo.doSomething();
        ...
    }
}
```

スコープ EJB クライアントコンテキストを使用するには、EJB クライアントプロパティーをプログラミングで設定し、コンテキスト作成でプロパティーを渡します。プロパティーは、標準的な **jboss-ejb-client.properties** ファイルで使用されるのと同じプロパティーセットです。EJB クライアントコンテキストを JNDI コンテキストにスコープ指定するには、**org.jboss.ejb.client.scoped.context** プロパティーを指定し、その値を **true** に設定する必要があります。このプロパティーは、EJB クライアント API に、EJB クライアントコンテキスト (EJB レシーバーにより支援される) を作成する必要があることと、作成されたコンテキストが作成元の JNDI コンテキストに対してのみスコープ指定されるか、可視状態であることを通知します。この JNDI コンテキストを使用してルックアップされた、または呼び出されたすべての EJB プロキシは、この JNDI コンテキストに関連付けられた EJB クライアントコンテキストのみを認識します。EJB をルックアップし、呼び出すためにアプリケーションにより使用される他の JNDI コンテキストは、他のスコープ EJB クライアントコンテキストについて認識しません。

org.jboss.ejb.client.scoped.context プロパティーを渡さず、EJB クライアントコンテキストにスコープ指定されない JNDI コンテキストはデフォルトの動作を使用します。デフォルトの動作では、一般的にアプリケーション全体に割り当てられた既存の EJB クライアントコンテキストを使用します。

スコープ EJB クライアントコンテキストは、JBoss Enterprise Application Platform の以前のバージョンの JNP ベース JNDI 呼び出しに関連する柔軟性をユーザーアプリケーションに提供します。スコープ EJB クライアントコンテキストにより、ユーザーアプリケーションはどの JNDI コンテキストがどのサーバーと通信するかや、JNDI コンテキストがどのようにサーバーと接続するかを細かく制御できるようになります。

バグを報告する

7.5.5. スコープ EJB クライアントコンテキストを使用した EJB の設定

概要

EJB は、マップベースのスコープコンテキストを使用して設定できます。これは、プログラムで、**jboss-ejb-client.properties** にある標準的なプロパティーを使用して **Properties** マップに値を入力し、**org.jboss.ejb.client.scoped.context** プロパティーに **true** を指定して、**InitialContext** のプロパティーを渡すことにより実現されます。

スコープコンテキストを使用する利点は、EJB を直接参照したり、JBoss クラスをインポートしたりせずにアクセスを設定できることです。また、マルチスレッド環境で実行時にホストを設定および負荷分散することが可能になります。

手順7.9 マップベースのスコープコンテキストを使用した EJB の設定

1. **プロパティーの設定**
   EJB クライアントプロパティーをプログラムで設定し、標準的な **jboss-ejb-client.properties** ファイルで使用されたのと同じプロパティーセットを指定します。ス

コープコンテキストを有効にするには、**org.jboss.ejb.client.scoped.context** プロパティーを指定し、その値を **true** に設定する必要があります。以下は、プロパティーをプログラムで設定する例です。

```
// Configure  EJB Client properties for the InitialContext
Properties ejbClientContextProps = new Properties();
ejbClientContextProps.put(“remote.connections”,”name1”);
ejbClientContextProps.put(“remote.connection.name1.host”,”localhost”
);
ejbClientContextProps.put(“remote.connection.name1.port”,”4447”);
// Property to enable scoped EJB client context which will be tied
to the JNDI context
ejbClientContextProps.put("org.jboss.ejb.client.scoped.context",
“true”);
```

2. **コンテキスト作成でプロパティーを渡す**

```
// Create the context using the configured properties
InitialContext ic = new InitialContext(ejbClientContextProps);
MySLSB bean = ic.lookup("ejb:myapp/ejb//MySLSBBean!" +
MySLSB.class.getName());
```

その他の情報

- ルックアップ EJB プロキシーにより生成されたコンテキストは、このスコープコンテキストによりバインドされ、重要な接続パラメーターのみを使用します。これにより、さまざまなコンテキストを作成してクライアントアプリケーション内のデータにアクセスしたり、さまざまなログインを使用してサーバーに独立してアクセスしたりできます。
- クライアントでは、スコープ **InitialContext** とスコーププロキシーの両方がスレッドに渡され、各スレッドが該当するコンテキストで動作することが可能になります。また、プロキシーを同時に使用できる複数のスレッドにプロキシーを渡すことができます。
- スコープコンテキスト EJB プロキシーは、リモートコールでシリアライズされ、サーバーでデシリアライズされます。デシリアライズされるとき、スコープコンテキスト情報が削除され、デフォルト状態に戻ります。デシリアライズされたプロキシーがリモートサーバーで使用される場合は、作成時に使用されたスコープコンテキストを持たなくなるため、**EJBCLIENT000025** エラーが発生したり、EJB 名を使用して間違った対象を呼び出したりすることがあります。

バグを報告する

7.5.6. EJB クライアントプロパティー

概要

以下の表は、プログラムまたは **jboss-ejb-client.properties** ファイルで設定できるプロパティーを示しています。

EJB クライアントグローバルプロパティー

以下の表は、同じスコープ内のライブラリー全体で有効なプロパティーを示しています。

表7.1 グローバルプロパティー

プロパティー名	説明
endpoint.name	クライアントエンドポイントの名前。設定されない場合、デフォルト値は **client-endpoint** です。 スレッド名にはこのプロパティーが含まれるため、異なるエンドポイント設定を区別するのに役に立つことがあります。
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED	すべての接続に対して SSL プロトコルが有効であるかどうかを指定するブール値。
deployment.node.selector	**org.jboss.ejb.client.DeploymentNodeSelector** の実装の完全修飾名。 これは、EJB の呼び出しを負荷分散するために使用されます。
invocation.timeout	EJB ハンドシェイクまたはメソッド呼び出し要求/応答サイクルのタイムアウト。この値はミリ秒単位です。 実行にタイムアウト時間よりも長い時間がかかった場合は、任意のメソッドの呼び出しで **java.util.concurrent.TimeoutException** がスローされます。実行が完了し、サーバーは中断されません。
reconnect.tasks.timeout	バックグラウンド再接続タスクのタイムアウト。この値はミリ秒単位です。 複数の接続がダウンしている場合は、次のクライアント EJB 呼び出しで、適切なノードを見つけるために再接続が必要かどうかを決定するアルゴリズムが使用されます。
org.jboss.ejb.client.scoped.context	スコープ EJB クライアントコンテキストを有効にするかどうかを指定するブール値。デフォルトは、**false** です。 **true** に設定された場合、EJB クライアントは JNDI コンテキストに割り当てられたスコープコンテキストを使用します。その他の場合、EJB クライアントコンテキストは JVM でグローバルセレクターを使用して、リモート EJB およびホストを呼び出すために使用されるプロパティーを決定します。

EJB クライアント接続プロパティー

接続プロパティーは、プレフィックス **remote.connection.*CONNECTION_NAME*** で始まります。*CONNECTION_NAME* は、接続を一意に識別するためにのみ使用されるローカル ID です。

表7.2 接続プロパティー

プロパティー名	説明
remote.connections	アクティブな **connection-names** のカンマ区切りのリスト。各接続はこの名前を使用して設定されます。
remote.connection.*CONNECTION_NAME*.host	接続のホスト名または IP。

プロパティー名	説明
remote.connection.*CONNECTION_NAME*.port	接続のポート。デフォルト値は **4447** です。
remote.connection.*CONNECTION_NAME*.username	接続セキュリティーを認証するために使用されるユーザー名。
remote.connection.*CONNECTION_NAME*.password	ユーザーを認証するために使用されるパスワード
remote.connection.*CONNECTION_NAME*.connect.timeout	初期接続のタイムアウト時間。この時間が経過すると、再接続タスクにより、接続を確立できるかどうかが定期的に確認されます。値はミリ秒単位です。
remote.connection.*CONNECTION_NAME*.callback.handler.class	**CallbackHandler** クラスの完全修飾名。これは、接続を確立するために使用され、接続がオープンである限り変更できません。
remote.connection.*CONNECTION_NAME*. **channel.options.org.jboss.remoting3.RemotingOptions.MAX_OUTBOUND_MESSAGES**	アウトバウンド要求の最大数を指定する整数値。デフォルト値は **80** です。 すべての呼び出しを処理するために、サーバーに対してクライアント **(JVM)** からの接続が **1** つだけあります。
remote.connection.*CONNECTION_NAME*. **connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS**	正常に接続するためにクライアントがクレデンシャルを提供する必要があるかどうかを決定するブール値。デフォルト値は **true** です。 **true** に設定された場合、クライアントはクレデンシャルを提供する必要があります。**false** に設定された場合は、リモートコネクターがセキュリティーレルムを要求しない限り、呼び出しが許可されます。
remote.connection.*CONNECTION_NAME*. **connect.options.org.xnio.Options.SASL_DISALLOWED_MECHANISMS**	接続作成中に認証に使用される特定の SASL メカニズムを無効にします。 **JBOSS_LOCAL_USER** の場合は、サイレント認証メカニズム **(**クライアントとサーバーが同じマシンにあるときに使用されます**)** が無効になります。

プロパティー名	説明
remote.connection.*CONNECTION_NAME*. **connect.options.org.xnio.Options.SASL_POLICY_NOPLAINTEXT**	認証中のプレーンテキストメッセージの使用を有効または無効にするブール値。JAAS を使用する場合は、プレーンテキストパスワードを許可するために false に設定する必要があります。
remote.connection.*CONNECTION_NAME*. **connect.options.org.xnio.Options.SSL_ENABLED**	この接続に対して SSL プロトコルが有効であるかどうかを指定するブール値。
remote.connection.*CONNECTION_NAME*. **connect.options.org.jboss.remoting3.RemotingOptions.HEARTBEAT_INTERVAL**	自動的なクローズ (ファイアウォールの場合など) を回避するためにクライアントとサーバー間でハートビートを送信する間隔。値はミリ秒単位です。

EJB クライアントクラスタープロパティー

初期接続でクラスター環境に接続する場合は、クラスターのトポロジーが自動的に非同期で受信されます。これらのプロパティーは、受信された各メンバーに接続するために使用されます。各プロパティーは、プレフィックス **remote.cluster.*CLUSTER_NAME*** で始まります。*CLUSTER_NAME* は、関連するサーバーの Infinispan サブシステム設定を参照します。

表7.3 クラスタープロパティ

プロパティー名	説明
remote.cluster.*CLUSTER_NAME*. **clusternode.selector**	**org.jboss.ejb.client.ClusterNodeSelector** の実装の完全修飾名。 このクラス (**org.jboss.ejb.clientDeploymentNodeSelector** ではない) は、クラスター環境で EJB 呼び出しを負荷分散するために使用されます。クラスターが完全にダウンしている場合、呼び出しは **No ejb receiver available** で失敗します。
remote.cluster.*CLUSTER_NAME*. **channel.options.org.jboss.remoting3.RemotingOptions.MAX_OUTBOUND_MESSAGES**	クラスター全体に対して実行できるアウトバウンド要求の最大数を指定する整数値。

プロパティー名	説明
remote.cluster.*CLUSTER_NAME*. **node.*NODE_NAME*.channel.options.org.jboss.remoting3.RemotingOptions.MAX_OUTBOUND_MESSAGES**	この特定のクラスターノードに対して実行できるアウトバウンド要求の最大数を指定する整数値。

バグを報告する

7.6. クラスター化された ENTERPRISE JAVABEANS

7.6.1. クラスター化された Enterprise JavaBean (EJB) について

高可用性が必要となる場合は、EJB コンポーネントをクラスター化することができます。EJB コンポーネントは 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、異なる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 とステートレス Bean をクラスター化できます。

シングルトンについては、「HA シングルトンの実装」を参照してください。

注記

EJB 2 エンティティー Bean は、クラスター化できません。この制限を変更する予定はありません。

バグを報告する

7.7. 参考資料

7.7.1. EJB JNDI の名前に関する参考資料

セッション Bean の JNDI ルックアップ名の構文は次の通りです。

```
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、EAR の名前になります。デフォルトでは、ファイル名から **.ear** サフィックスを除いたものが EAR の名前になります。また、アプリケーション名を **application.xml** ファイルで上書きすることも可能です。セッション Bean が EAR にデプロイされていない場合は空白のままにしておきます。

<moduleName>

モジュール名はセッション Bean がデプロイされた JAR ファイルの名前になります。デフォルトでは、ファイル名から **.jar** サフィックスを除いたものが JAR ファイルの名前になります。また、モジュール名を JAR の **ejb-jar.xml** ファイルで上書きすることも可能です。

<distinctName>

JBoss Enterprise Application Platform 6 では、各デプロイメントが任意の個別名を指定することができます。デプロイメントの個別名がない場合は空白のままにしておきます。

<beanName>

Bean 名は呼び出されるセッション Bean のクラス名です。

<viewClassName>

ビュークラス名はリモートインターフェースの完全修飾クラス名です。インターフェースのパッケージ名が含まれます。

?stateful

JNDI 名がステートフルセッション Bean を参照する時に **?stateful** サフィックスが必要となります。他の Bean タイプでは含まれていません。

バグを報告する

7.7.2. EJB 参照の解決

本項では、JBoss が **@EJB** や **@Resource** を実装する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** アノテーションは **mappedName()** 属性を持っています。仕様はこのベンダー固有のメタデータを無視しますが、 JBoss は参照しているEJBのグローバル JNDI 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、このグローバル JNDI 名がバインディングに使用されます。
- 以下のように属性を定義せずに **@EJB** を指定するとします。

  ```
  @EJB
  ProcessPayment myEjbref;
  ```

  この場合、次のルールが適用されます。

  - 参照する Bean の EJB jar が、**@EJB** 挿入に使用されるインターフェースを持つ EJB に対して検索されます。同じビジネスインターフェースをパブリッシュする EJB が複数ある場合、例外がスローされます。インターフェースを持つ Bean が 1 つのみである場合はその Bean が使用されます。
  - そのインターフェースをパブリッシュする EJB に対する EAR を検索します。複製がある場合は例外がスローされます。それ以外の場合は、一致する Bean が返されます。
  - JBoss ランタイムでそのインターフェースの EJB に対してグローバルに検索が行われます。ここでも複製があると例外がスローされます。
- **@EJB.beanName()** は **<ejb-link>** に対応します。**beanName()** が定義されている場合、属性が定義されていない **@EJB** として同じアルゴリズムが使用されますが、検索で**beanName()** がキーとして使用されます。ejb-link の # 構文を使用する場合、このルールの例外となります。# 構文は、参照する EJB が存在する EAR の jar への相対パスを指定できるようにします。詳細については EJB 3.1 仕様を参照してください。

バグを報告する

7.7.3. 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 プロジェクトには JBoss Enterprise Application Platform 6 の Maven リポジトリより次の依存関係が必要となります。

表7.4 リモート EJB クライアントに対する Maven の依存関係

GroupID	ArtifactID
org.jboss.spec	jboss-javaee-6.0
org.jboss.as	jboss-as-ejb-client-bom
org.jboss.spec.javax.transaction	jboss-transaction-api_1.1_spec
org.jboss.spec.javax.ejb	jboss-ejb-api_3.1_spec
org.jboss	jboss-ejb-client
org.jboss.xnio	xnio-api
org.jboss.xnio	xnio-nio
org.jboss.remoting3	jboss-remoting
org.jboss.sasl	jboss-sasl
org.jboss.marshalling	jboss-marshalling-river

jboss-javaee-6.0 と **jboss-as-ejb-client-bom** を除き、これらの依存関係を **pom.xml** ファイルの **<dependencies>** セクションに追加する必要があります。

jboss-javaee-6.0 と **jboss-as-ejb-client-bom** の依存関係は、スコープが **import** の **pom.xml** の **<dependencyManagement>** セクションに追加する必要があります。

注記

artifactID のバージョンは変更される可能性があります。該当バージョンについては、Maven リポジトリーを参照してください。

```
<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.jboss.spec</groupId>
         <artifactId>jboss-javaee-6.0</artifactId>
         <version>3.0.0.Final-redhat-1</version>
         <type>pom</type>
         <scope>import</scope>
```

```
        </dependency>

        <dependency>
           <groupId>org.jboss.as</groupId>
           <artifactId>jboss-as-ejb-client-bom</artifactId>
           <version>7.1.1.Final-redhat-1</version>
           <type>pom</type>
           <scope>import</scope>
        </dependency>
     </dependencies>
  </dependencyManagement>
```

リモートセッション Bean の呼び出しに対する依存関係設定の例は **remote-ejb/client/pom.xml** を参照してください。

バグを報告する

7.7.4. jboss-ejb3.xml デプロイメント記述子に関する参考文書

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の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、WAR アーカイブでは **WEB-INF/** ディレクトリにある必要があります。

形式は **ejb-jar.xml** に似ていて、同じ名前空間を一部使用し、他の名前空間を一部提供します。**jboss-ejb3.xml** の内容は **ejb-jar.xml** の内容と結合されますが、**jboss-ejb3.xml** の項目の方が優先されます。

本文書では **jboss-ejb3.xml** によって使用される非標準の名前空間のみ取り上げます。標準的な名前空間については http://java.sun.com/xml/ns/javaee/ のドキュメント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。

アセンブリ記述子の名前空間

次の名前空間はすべて **<assembly-descriptor>** 要素で使用されます。これらの名前空間の設定を 1 つの Bean に適用したり、***** を **ejb-name** として使用してデプロイメントのすべての Bean に対して適用するために使用されます。

クラスタリング名前空間:urn:clustering:1.0

```
xmlns:c="urn:clustering:1.0"
```

これにより、EJB がクラスター化されているとマーク付けすることができます。これは **@org.jboss.ejb3.annotation.Clustered** に相当するデプロイメント記述子です。

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

セキュリティー名前空間 (urn:security)

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

これにより、EJB のセキュリティードメインと run-as プリンシパルを設定できます。

```
<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 に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>myResourceAdaptor</r:resource-adapter-name>
</r:resource-adapter-binding>
```

IIOP 名前空間: urn:iiop

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

IIOP 名前空間には IIOP が設定されます。

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 やメッセージ駆動 Bean によって使用され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>
```

例7.1 jboss-ejb3.xml ファイルの例

```
<?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:1.0"

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">
      <enterprise-beans>
         <message-driven>
            <ejb-name>ReplyingMDB</ejb-name>
            <ejb-
class>org.jboss.as.test.integration.ejb.mdb.messagedestination.ReplyingM
DB</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>
      <assembly-descriptor>
         <c:clustering>
            <ejb-name>DDBasedClusteredSFSB</ejb-name>
            <c:clustered>true</c:clustered>
         </c:clustering>
      </assembly-descriptor>
   </jboss:ejb-jar>
```

バグを報告する

第8章 WEB アプリケーションのクラスター化

8.1. セッションレプリケーション

8.1.1. HTTP セッションレプリケーションについて

セッションレプリケーションにより、分散可能なアプリケーションのクライアントセッションがクラスター内のノードのフェイルオーバーなどで中断されないようにします。クラスター内の各ノードは実行中のセッションの情報を共有するため、もともと関連していたノードが消えた場合も作業を引き継ぐことができます。

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

バグを報告する

8.1.2. Web セッションキャッシュについて

Web セッションキャッシュは、**standalone-ha.xml** プロファイルを含むいずれかの HA プロファイル、管理対象ドメインプロファイル **ha** または **full-ha** を使用するときに設定できます。最も一般的に設定される要素は、キャッシュモードと分散キャッシュのキャッシュオーナーの数です。

キャッシュモード

キャッシュモードは、**REPL** (デフォルト値) または **DIST** のいずれかになります。

REPL

REPL モードでは、クラスターの他のノードそれぞれにキャッシュ全体がレプリケートされます。これは、最も安全なオプションですが、オーバーヘッドが増加します。

DIST

DIST モードは、以前の実装で提供された*バディモード*に似ています。このモードでは、**owners** パラメーターで指定された数のノードにキャッシュを分散することによりオーバーヘッドが削減されます。オーナーのこの数のデフォルト値は **2** です。

オーナー

owners パラメーターは、セッションのレプリケートされたコピーを保持するクラスターノード数を制御します。デフォルト値は、**2** です。

バグを報告する

8.1.3. Web セッションキャッシュの設定

Web セッションキャッシュのデフォルト値は **REPL** です。**DIST** モードを使用する場合は、管理 CLI で次の 2 つのコマンドを実行します。異なるプロファイルを使用する場合は、コマンドでプロファイル名を変更します。スタンドアロンサーバーを使用する場合は、コマンドの **/profile=ha** 部分を削除します。

手順8.1 Web セッションキャッシュの設定

1. **デフォルトキャッシュモードを DIST に変更します。**

```
/profile=ha/subsystem=infinispan/cache-container=web/:write-
attribute(name=default-cache,value=dist)
```

2. **分散キャッシュのオーナー数を設定します。**
   以下のコマンドでは、**5** オーナーが設定されます。デフォルト値は **2** です。

```
/profile=ha/subsystem=infinispan/cache-container=web/distributed-
cache=dist/:write-attribute(name=owners,value=5)
```

3. **デフォルトキャッシュモードを REPL に戻します。**

```
/profile=ha/subsystem=infinispan/cache-container=web/:write-
attribute(name=default-cache,value=repl)
```

4. **サーバーの再起動**
   Web キャッシュモードの変更後は、サーバーを再起動する必要があります。

結果

サーバーでセッションレプリケーションが設定されます。独自のアプリケーションでセッションレプリケーションを使用するには、「アプリケーションにおけるセッションレプリケーションの有効化」を参照してください。

バグを報告する

8.1.4. アプリケーションにおけるセッションレプリケーションの有効化

概要

JBoss Enterprise Application Platform の高可用性 (HA) 機能を利用するには、アプリケーションが配布可能になるよう設定する必要があります。ここでは配布可能にする手順を説明した後、使用可能な高度な設定オプションの一部について解説します。

手順8.2 アプリケーションを配布可能にする

1. **要件: アプリケーションが配布可能であることを示します。**
   アプリケーションが配布可能とマークされていないとセッションが配布されません。アプリケーションの **web.xml** 記述子ファイルの **<web-app>** タグ内に **<distributable />** 要素を追加します。例は次の通りです。

例8.1 配布可能なアプリケーションの最低限の設定

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

http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
          version="2.4">

      <distributable/>
```

```
</web-app>
```

2. **希望する場合はデフォルトのレプリケーション動作を変更します。**
   セッションレプリケーションに影響する値を変更したい場合は、**<jboss-web>** 要素の子要素である **<replication-config>** 要素内で値を上書きします。デフォルトを上書きしたい場合のみ指定の要素が含まれるようにします。以下の例に、全デフォルト設定の一覧と、最も一般的に変更されるオプションを説明する表を示します。

例8.2 デフォルトの <replication-config> 値

```
<!DOCTYPE jboss-web PUBLIC
    "-//JBoss//DTD Web Application 5.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss-web_5_0.dtd">

<jboss-web>

   <replication-config>
      <cache-name>custom-session-cache</cache-name>
      <replication-trigger>SET</replication-trigger>
      <replication-granularity>ATTRIBUTE</replication-
granularity>
      <use-jk>false</use-jk>
      <max-unreplicated-interval>30</max-unreplicated-interval>
      <snapshot-mode>INSTANT</snapshot-mode>
      <snapshot-interval>1000</snapshot-interval>
      <session-notification-
policy>com.example.CustomSessionNotificationPolicy</session-
notification-policy>
  </replication-config>

</jboss-web>
```

表8.1 セッションレプリケーションの一般的なオプション

オプション	説明

オプション	説明
<replication-trigger>	クラスター全体でセッションデータのレプリケーションが引き起こされるのはどのような状態であるか制御します。セッション属性として保存された可変オブジェクトがセッションからアクセスされた後、メソッド **setAttribute()** が直接呼び出されない限り、オブジェクトが変更されレプリケーションが必要であるかをコンテナは明確に認識できないため、このオプションは必須となります。 **<replication-trigger>の有効な値** **SET_AND_GET** 最も安全で、最もパフォーマンスが悪いオプションになります。コンテンツへのアクセスのみが行われ、変更されなくても常にセッションデーターがレプリケートされます。この設定はレガシー機能に対応する目的でのみ保持されています。同じ動作のパフォーマンスを向上させるには、この設定を使用する代わりに、**<max_unreplicated_interval>** を 0 に設定します。 **SET_AND_NON_PRIMITIVE_GET** デフォルト値です。非プリミティブ型のオブジェクトがアクセスされた時のみセッションデータがレプリケートされます。オブジェクトは **Integer**、**Long**、**String** などの良く知られた Java データ型ではありません。 **SET** このオプションは、データのレプリケーションが必要な時にセッション上でアプリケーションが **setAttribute** を明示的に呼び出すことを前提としています。これにより、不必要なレプリケーションの発生を防ぎ、全体的なパフォーマンスも改善されますが、本質的に安全ではありません。 設定に関係なく、**setAttribute()** を呼び出すと常にセッションレプリケーションが引き起こされます。
<replication-granularity>	レプリケートされるデータの細かさを決定します。デフォルトは **SESSION** ですが、**ATTRIBUTE** を設定すると、ほとんどの属性は変更されずにセッションのパフォーマンスを向上することができます。

以下は変更する必要がほとんどないオプションになります。

表8.2 セッションレプリケーションの変更が稀なオプション

オプション	説明
<useJK>	**mod_cluster**、**mod_jk**、**mod_proxy** などのロードバランサーの使用を前提とするか指定します。デフォルトは **false** です。**true** に設定すると、各要求に関連付けられているセッション ID がコンテナによって確認され、フェイルオーバーが発生するとセッション ID の **jvmRoute** の部分が置き換えられます。

オプション	説明
<max-unreplicated-interval>	セッションのタイムスタンプのレプリケーションがトリガーされるまで、セッション後に待機する最大間隔 (秒単位) になります。変更がないと判断された場合でも適用されます。これにより、各セッションのタイムスタンプがクラスターノードによって認識されるようにし、フェイルオーバー中にレプリケートされなかったセッションが誤って期限切れにならないようにします。また、フェイルオーバー中に **HttpSession.getLastAccessedTime()** への呼び出しに正しい値を使用できるようにします。 デフォルトでは値は指定されません。値が指定されないと、コンテナの **jvmRoute** 設定が JK フェイルオーバーが使用されているかを判断します。 **0** を設定すると、セッションがアクセスされるたびにタイムスタンプがレプリケートされます。 **-1** を設定すると、要求中の他のアクティビティがレプリケーションをトリガーした場合のみタイムスタンプがレプリケートされます。 **HttpSession.getMaxInactiveInterval()** よりも大きい正の値を設定すると設定ミスとして扱われ、 **0** に変換されます。
<snapshot-mode>	セッションが他のノードへレプリケートされるタイミングを指定します。デフォルトは **INSTANT** で、 **INTERVAL** を使用することも可能です。 **INSTANT** モードでは要求処理スレッドが使用され、変更は要求の最後にレプリケートされます。 **<snapshot-interval>** オプションは無視されます。 **INTERVAL** モードでは、バックグラウンドタスクは **<snapshot-interval>** によって指定される間隔で実行され、変更されたセッションがレプリケートされます。
<snapshot-interval>	**INTERVAL** が **<snapshot-mode>** の値として使用された時に、変更されたセッションがレプリケートされる間隔 (ミリ秒単位) になります。
<session-notification-policy>	インターフェース **ClusteredSessionNotificationPolicy** の実装の完全修飾クラス名です。登録された **HttpSessionListener**、 **HttpSessionAttributeListener**、 **HttpSessionBindingListener** へサーブレット仕様の通知が送信されたかどうかを管理します。

バグを報告する

8.2. HTTPSESSION の非活性化および活性化

8.2.1. HTTP セッションパッシベーションおよびアクティベーション

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

*アクティベーション*とは、パッシベートされたデータを永続ストレージから読み出し、メモリに戻すことを言います。

パッシベーションは HTTP セッションのライフタイムで 3 回発生します。

- コンテナが新規セッションの作成を要求する時に現在アクティブなセッションの数が設定上限を越えている場合、サーバーはセッションの一部をパッシベートして新規セッションを作成できるようにします。
- 設定された間隔で、定期的にバックグラウンドタスクがセッションをパッシベートすべきかチェックします。
- ある Web アプリケーションがデプロイされ、他のサーバーでアクティブなセッションのバックアップコピーが、新たにデプロイする Web アプリケーションのセッションマネージャーによって取得された場合、セッションはパッシベートされることがあります。

以下の条件を満たすとセッションはパッシベートされます。

- セッションが設定した最大アイドル時間の間利用されていない。
- アクティブなセッションの数が設定上限を越えず、セッションがアイドル時間の設定下限を超えていない。

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

バグを報告する

8.2.2. アプリケーションにおける HttpSession パッシベーションの設定

概要

HttpSession パッシベーションはアプリケーションの **WEB_INF/jboss-web.xml** ファイルまたは **META_INF/jboss-web.xml** ファイルで設定されます。

例8.3 jboss-web.xml ファイルの例

```
<!DOCTYPE jboss-web PUBLIC
    "-//JBoss//DTD Web Application 5.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss-web_5_0.dtd">

<jboss-web version="6.0"
           xmlns="http://www.jboss.com/xml/ns/javaee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-web_6_0.xsd">

   <max-active-sessions>20</max-active-sessions>
   <passivation-config>
      <use-session-passivation>true</use-session-passivation>
      <passivation-min-idle-time>60</passivation-min-idle-time>
      <passivation-max-idle-time>600</passivation-max-idle-time>
   </passivation-config>


</jboss-web>
```

パッシベーション設定要素

<max-active-sessions>

許可されるアクティブセッションの最大数です。パッシベーションが有効になっている場合、セッションマネージャーによって管理されるセッション数がこの値を越えると、設定された **<passivation-min-idle-time>** を基に過剰なセッションがパッシベートされます。それでもアクティブセッションの数が制限を越える場合は、新しいセッションの作成に失敗します。デフォルト値は **-1** で、アクティブセッションの最大数は制限されません。

<passivation-config>

この要素は、子要素などの残りのパッシベーション設定パラメーターを保持します。

<passivation-config> 子要素

<use-session-passivation>

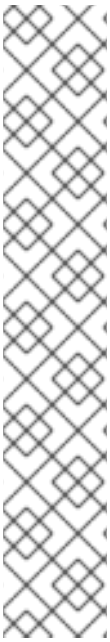
セッションパッシベーションを使用するかどうか。デフォルト値は **false** です。

<passivation-min-idle-time>

アクティブなセッションの数を減らし max-active-sessions によって定義された値に従うため、コンテナがパッシベーションの実行を考慮する前にセッションが非アクティブでなければならない最小期間。デフォルト値は **-1** で、**<passivation-max-idle-time>** が経過する前のセッションのパッシベートを無効にします。**<max-active-sessions>** が設定されている場合、-1 や大きな値は推奨されません。

<passivation-max-idle-time>

メモリーを節約するため、コンテナがパッシベーションを実行しようとする前にセッションが非アクティブにならなければならない最大期間。アクティブセッションの数が **<max-active-sessions>** を越えるかどうかに関係なく、このようなセッションのパッシベーションは実行されます。この値は **web.xml** の **<session-timeout>** 設定よりも小さい値とする必要があります。デフォルト値は **-1** で、非アクティブとなる最大期間を基にしたパッシベーションを無効にします。



注記

メモリーのセッション合計数にはこのノードでアクセスされていない他のクラスターノードからレプリケートされたセッションが含まれています。これを考慮して **<max-active-sessions>** を設定してください。また、他のノードからレプリケートされるセッションの数は、**REPL** または **DIST** キャッシュモードが有効であるかどうかによっても異なります。**REPL** キャッシュモードでは、各セッションは各ノードにレプリケートされます。**DIST** キャッシュモードでは、各セッションは、**owner** パラメーターによって指定された数のノードにのみレプリケートされます。セッションキャッシュモードの設定については、「Web セッションキャッシュについて」および「Web セッションキャッシュの設定」を参照してください。

たとえば、各ノードが 100 人のユーザーからの要求を処理する 8 つのノードを持つクラスターについて考えてみましょう。**REPL** キャッシュモードでは、各ノードはメモリーに 800 のセッションを保存します。**DIST** キャッシュモードが有効であり、デフォルトの **owners** 設定が **2** である場合、各ノードはメモリーに 200 のセッションを保存します。

バグを報告する

8.3. クッキードメイン

8.3.1. クッキードメインについて

*クッキードメイン*とは、アプリケーションにアクセスしているクライアントブラウザーからクッキーを読み取ることができるホストのセットのことです。アプリケーションがブラウザークッキーに保存する情報へ第三者がアクセスするリスクを最小限にする設定メカニズムになります。

クッキードメインのデフォルト値は / です。これは、発行ホストのみがクッキーの内容を読み取ることができます。特定のクッキードメインを設定すると、さまざまなホストがクッキーの内容を読み取ることができるようになります。クッキードメインの設定は「クッキードメインの設定」を参照してください。

バグを報告する

8.3.2. クッキードメインの設定

SSO コンテキストを共有するため SSO バルブを有効にするには、バルブ設定のクッキードメインを設定します。次の設定は、関連するクラスターや仮想ホストの異なるサーバーで実行されるアプリケーションが複数のエイリアスを持つ場合でも、**http://app1.xyz.com** および **http://app2.xyz.com** 上のアプリケーションが SSO コンテキストを共有できるようにします。

例8.4 クッキードメインの設定例

```
<Valve
className="org.jboss.web.tomcat.service.sso.ClusteredSingleSignOn"
                cookieDomain="xyz.com" />
```

バグを報告する

8.4. HA シングルトンの実装

概要

JBoss Enterprise Application Platform 5 では、HA シングルトンアーカイブは他のデプロイメントとは別に **deploy-hasingleton/** ディレクトリーにデプロイされていました。これは自動デプロイメントが発生しないようにするためで、また確実に HASingletonDeployer サービスがデプロイメントを制御し、クラスターのマスターノードのみにアーカイブがデプロイされるようにするための処置でした。ホットデプロイメント機能がなかったため、再デプロイメントにはサーバーの再起動が必要でした。また、マスターノードに障害が発生し、他のノードがマスターとして引き継ぐ必要がある場合、シングルトンサービスはサービスを提供するためデプロイメントプロセス全体を実行する必要がありました。

JBoss Enterprise Application Platform 6 ではこれが変更になりました。SingletonService を使用してクラスターの各ノードに目的のサービスがインストールされますが、サービスは一度に 1 つのノード上でのみ起動されます。これにより、デプロイメントの要件が簡素化され、ノード間でシングルトンマスターサービスを移動するために必要な時間が最小限になります。

手順8.3 HA シングルトンサービスの実装

1. **HA シングルトンサービスアプリケーションを作成します。**
   シングルトンサービスとしてデプロイされる SingletonService デコレーターでラッピングされたサービスの簡単な例は次のとおりです。

a. **シングルトンサービスを作成します。**
以下のリストは、シングルトンサービスの例です。

```
package com.mycompany.hasingleton.service.ejb;

import java.util.concurrent.atomic.AtomicBoolean;
import java.util.logging.Logger;

import org.jboss.as.server.ServerEnvironment;
import org.jboss.msc.inject.Injector;
import org.jboss.msc.service.Service;
import org.jboss.msc.service.ServiceName;
import org.jboss.msc.service.StartContext;
import org.jboss.msc.service.StartException;
import org.jboss.msc.service.StopContext;
import org.jboss.msc.value.InjectedValue;

/**
 * @author <a href="mailto:wfink@redhat.com">Wolf-Dieter Fink</a>
 */
public class EnvironmentService implements Service<String> {
    private static final Logger LOGGER =
Logger.getLogger(EnvironmentService.class.getCanonicalName());
    public static final ServiceName SINGLETON_SERVICE_NAME =
ServiceName.JBOSS.append("quickstart", "ha", "singleton");
    /**
     * A flag whether the service is started.
     */
    private final AtomicBoolean started = new
AtomicBoolean(false);

    private String nodeName;

    private final InjectedValue<ServerEnvironment> env = new
InjectedValue<ServerEnvironment>();

    public Injector<ServerEnvironment> getEnvInjector() {
        return this.env;
    }

    /**
     * @return the name of the server node
     */
    public String getValue() throws IllegalStateException,
IllegalArgumentException {
        if (!started.get()) {
            throw new IllegalStateException("The service '" +
this.getClass().getName() + "' is not ready!");
        }
        return this.nodeName;
    }

    public void start(StartContext arg0) throws StartException {
        if (!started.compareAndSet(false, true)) {
            throw new StartException("The service is still
started!");
```

```
        }
        LOGGER.info("Start service '" + this.getClass().getName()
+ "'");
        this.nodeName = this.env.getValue().getNodeName();
    }

    public void stop(StopContext arg0) {
        if (!started.compareAndSet(true, false)) {
            LOGGER.warning("The service '" +
this.getClass().getName() + "' is not active!");
        } else {
            LOGGER.info("Stop service '" +
this.getClass().getName() + "'");
        }
    }
}
```

b. **サーバー起動時にサービスを SingletonService として起動するためにシングルトン EJB を作成します。**
以下のリストは、サーバー起動時に SingletonService を起動するシングルトン EJB の例です。

```
package com.mycompany.hasingleton.service.ejb;

import java.util.Collection;
import java.util.EnumSet;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Singleton;
import javax.ejb.Startup;

import org.jboss.as.clustering.singleton.SingletonService;
import org.jboss.as.server.CurrentServiceContainer;
import org.jboss.as.server.ServerEnvironment;
import org.jboss.as.server.ServerEnvironmentService;
import org.jboss.msc.service.AbstractServiceListener;
import org.jboss.msc.service.ServiceController;
import org.jboss.msc.service.ServiceController.Transition;
import org.jboss.msc.service.ServiceListener;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


/**
 * A Singleton EJB to create the SingletonService during startup.
 *
 * @author <a href="mailto:wfink@redhat.com">Wolf-Dieter Fink</a>
 */
@Singleton
@Startup
public class StartupSingleton {
  private static final Logger LOGGER =
LoggerFactory.getLogger(StartupSingleton.class);
```

```
  /**
   * Create the Service and wait until it is started.<br/>
   * Will log a message if the service will not start in 10sec.
   */
  @PostConstruct
  protected void startup() {
    LOGGER.info("StartupSingleton will be initialized!");

    EnvironmentService service = new EnvironmentService();
    SingletonService<String> singleton = new
SingletonService<String>(service,
EnvironmentService.SINGLETON_SERVICE_NAME);
    // if there is a node where the Singleton should deployed the
election policy might set,
    // otherwise the JGroups coordinator will start it
    //singleton.setElectionPolicy(new
PreferredSingletonElectionPolicy(new
NamePreference("node2/cluster"), new
SimpleSingletonElectionPolicy()));
    ServiceController<String> controller =
singleton.build(CurrentServiceContainer.getServiceContainer())
        .addDependency(ServerEnvironmentService.SERVICE_NAME,
ServerEnvironment.class, service.getEnvInjector())
        .install();

    controller.setMode(ServiceController.Mode.ACTIVE);
    try {
      wait(controller, EnumSet.of(ServiceController.State.DOWN,
ServiceController.State.STARTING), ServiceController.State.UP);
      LOGGER.info("StartupSingleton has started the Service");
    } catch (IllegalStateException e) {
      LOGGER.warn("Singleton Service {} not started, are you sure
to start in a cluster (HA)
environment?",EnvironmentService.SINGLETON_SERVICE_NAME);
    }
  }

  /**
   * Remove the service during undeploy or shutdown
   */
  @PreDestroy
  protected void destroy() {
    LOGGER.info("StartupSingleton will be removed!");
    ServiceController<?> controller =
CurrentServiceContainer.getServiceContainer().getRequiredService(
EnvironmentService.SINGLETON_SERVICE_NAME);
    controller.setMode(ServiceController.Mode.REMOVE);
    try {
      wait(controller, EnumSet.of(ServiceController.State.UP,
ServiceController.State.STOPPING, ServiceController.State.DOWN),
ServiceController.State.REMOVED);
    } catch (IllegalStateException e) {
      LOGGER.warn("Singleton Service {} has not be stopped
correctly!",EnvironmentService.SINGLETON_SERVICE_NAME);
    }
  }
```

```
  private static <T> void wait(ServiceController<T> controller,
Collection<ServiceController.State> expectedStates,
ServiceController.State targetState) {
    if (controller.getState() != targetState) {
      ServiceListener<T> listener = new
NotifyingServiceListener<T>();
      controller.addListener(listener);
      try {
        synchronized (controller) {
          int maxRetry = 2;
          while (expectedStates.contains(controller.getState())
&& maxRetry > 0) {
            LOGGER.info("Service controller state is {}, waiting
for transition to {}", new Object[] {controller.getState(),
targetState});
            controller.wait(5000);
            maxRetry--;
          }
        }
      } catch (InterruptedException e) {
        LOGGER.warn("Wait on startup is interrupted!");
        Thread.currentThread().interrupt();
      }
      controller.removeListener(listener);
      ServiceController.State state = controller.getState();
      LOGGER.info("Service controller state is now {}",state);
      if (state != targetState) {
        throw new IllegalStateException(String.format("Failed to
wait for state to transition to %s.  Current state is %s",
targetState, state), controller.getStartException());
      }
    }
  }

  private static class NotifyingServiceListener<T> extends
AbstractServiceListener<T> {
    @Override
    public void transition(ServiceController<? extends T>
controller, Transition transition) {
      synchronized (controller) {
        controller.notify();
      }
    }
  }
}
```

c. **クライアントよりサービスへアクセスするためステートレスセッション Bean を作成します。**
以下は、クライアントからサービスにアクセスするステートレスセッション Bean の例です。

```
package com.mycompany.hasingleton.service.ejb;

import javax.ejb.Stateless;
```

```
import org.jboss.as.server.CurrentServiceContainer;
import org.jboss.msc.service.ServiceController;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * A simple SLSB to access the internal SingletonService.
 *
 * @author <a href="mailto:wfink@redhat.com">Wolf-Dieter Fink</a>
 */
@Stateless
public class ServiceAccessBean implements ServiceAccess {
    private static final Logger LOGGER =
LoggerFactory.getLogger(ServiceAccessBean.class);

    public String getNodeNameOfService() {
        LOGGER.info("getNodeNameOfService() is called()");
        ServiceController<?> service =
CurrentServiceContainer.getServiceContainer().getService(
                EnvironmentService.SINGLETON_SERVICE_NAME);
        LOGGER.debug("SERVICE {}", service);
        if (service != null) {
            return (String) service.getValue();
        } else {
            throw new IllegalStateException("Service '" +
EnvironmentService.SINGLETON_SERVICE_NAME + "' not found!");
        }
    }
}
```

d. **SingletonService のビジネスロジックインターフェースを作成します。**
以下は、SingletonService に対するビジネスロジックインターフェースの例です。

```
package com.mycompany.hasingleton.service.ejb;

import javax.ejb.Remote;

/**
 * Business interface to access the SingletonService via this EJB
 *
 * @author <a href="mailto:wfink@redhat.com">Wolf-Dieter Fink</a>
 */
@Remote
public interface ServiceAccess {
    public abstract String getNodeNameOfService();
}
```

2. **クラスタリングが有効な状態で各 Jboss Enterprise Application Platform 6 インスタンスを起動します。**

   クラスターを有効化する方法は、サーバーがスタンドアロンであるか管理ドメインで実行されているかによって異なります。

a. **管理対象ドメインで実行されているサーバーに対してクラスタリングを有効にします。**
管理 CLI を使用してクラスタリングを有効にするか、設定ファイルを手動で編集できます。

- **管理 CLI を使用してクラスタリングを有効にします。**

i. **ドメインコントローラーを起動します。**

ii. **使用しているオペレーティングシステムのコマンドプロンプトを開きます。**

iii. **ドメインコントローラーの IP アドレスまたは DNS 名を渡して管理 CLI に接続します。**
この例では、ドメインコントローラーの IP アドレスが **192.168.0.14** であることを前提とします。

- Linux の場合は、コマンドラインで以下を入力します。

```
$ EAP_HOME/bin/jboss-cli.sh --connect --
controller=192.168.0.14
$ Connected to domain controller at 192.168.0.14
```

- Windows の場合は、コマンドラインで以下を入力します。

```
C:\>EAP_HOME\bin\jboss-cli.bat --connect --
controller=192.168.0.14
C:\> Connected to domain controller at 192.168.0.14
```

iv. **main-server サーバーグループを追加します。**

```
[domain@192.168.0.14:9999 /] /server-group=main-server-
group:add(profile="ha",socket-binding-group="ha-sockets")
{
    "outcome" => "success",
    "result" => undefined,
    "server-groups" => undefined
}
```

v. **server-one という名前のサーバーを作成し、main-server サーバーグループに追加します。**

```
[domain@192.168.0.14:9999 /]  /host=station14Host2/server-
config=server-one:add(group=main-server-group,auto-
start=false)
{
    "outcome" => "success",
    "result" => undefined
}
```

vi. **main-server サーバーグループに対して JVM を設定します。**

```
[domain@192.168.0.14:9999 /] /server-group=main-server-
group/jvm=default:add(heap-size=64m,max-heap-size=512m)
{
    "outcome" => "success",
```

```
    "result" => undefined,
    "server-groups" => undefined
}
```

vii. **server-two という名前のサーバーを作成し、別のサーバーグループに置き、ポートオフセットを 100 に設定します。**

```
[domain@192.168.0.14:9999 /]  /host=station14Host2/server-
config=server-two:add(group=distinct2,socket-binding-port-
offset=100)
{
    "outcome" => "success",
    "result" => undefined
}
```

- **サーバー設定ファイルを手動で編集してクラスタリングを有効にします。**

  i. **JBoss Enterprise Application Platform サーバーを停止します。**

  重要

  変更がサーバーの再起動後にも維持されるようにするには、サーバー設定ファイルの編集前にサーバーを停止する必要があります。

  ii. **domain.xml 設定ファイルを開いて編集します。**
  ha プロファイルと **ha-sockets** ソケットバインディンググループを使用するサーバーグループを指定します。例は次のとおりです。

```
<server-groups>
  <server-group name="main-server-group" profile="ha">
    <jvm name="default">
      <heap size="64m" max-size="512m"/>
    </jvm>
    <socket-binding-group ref="ha-sockets"/>
  </server-group>
</server-groups>
```

  iii. **host.xml 設定ファイルを開いて編集します。**
  以下のようにファイルを変更します。

```
<servers>
  <server name="server-one" group="main-server-group" auto-
start="false"/>
  <server name="server-two" group="distinct2">
    <socket-bindings port-offset="100"/>
  </server>
<servers>
```

  iv. **サーバーを起動します。**

  - Linux の場合は、***EAP_HOME*/bin/domain.sh** と入力します。

- Microsoft Windows の場合は、***EAP_HOME*\bin\domain.bat** と入力します。

b. **スタンドアロンサーバーに対してクラスタリングを有効にします。**
スタンドアロンサーバーに対してクラスタリングを有効にするには、次のようにノード名と **standalone-ha.xml** 設定ファイルを使用してサーバーを起動します。

- Linux の場合は、***EAP_HOME*/bin/standalone.sh --server-config=standalone-ha.xml -Djboss.node.name=*UNIQUE_NODE_NAME*** と入力します。
- Microsoft Windows の場合は、***EAP_HOME*\bin\standalone.bat --server-config=standalone-ha.xml -Djboss.node.name=*UNIQUE_NODE_NAME*** と入力します。

注記

1 つのマシン上で複数のサーバーが実行されている時にポートの競合が発生しないようにするため、別のインターフェースでバインドするように各サーバーインスタンスに対して **standalone-ha.xml** ファイルを設定します。または、コマンドラインで **-Djboss.socket.binding.port-offset=100** のような引数を使用し、ポートオフセットを持つ後続のサーバーインスタンスを開始して対応することも可能です 。

3. **アプリケーションをサーバーにデプロイします。**
Maven を使用してアプリケーションをデプロイする場合は、次の Maven コマンドを使用してデフォルトのポートで稼働しているサーバーへデプロイします。

```
mvn clean install jboss-as:deploy
```

追加のサーバーをデプロイするには、サーバー名とポート番号をコマンドラインに渡します。

```
mvn clean package jboss-as:deploy -Ddeploy.hostname=localhost -
Ddeploy.port=10099
```

バグを報告する

第9章 CDI

9.1. CDI の概要

9.1.1. CDI の概要

- 「Contexts and Dependency Injection (CDI) について」
- 「Weld、Seam 2、 および JavaServer Faces 間の関係」
- 「CDI の利点」

バグを報告する

9.1.2. Contexts and Dependency Injection (CDI) について

Contexts and Dependency Injection (CDI) は、EJB 3.0 コンポーネントを Java Server Faces (JSF) 管理対象 Bean として使用できるよう設計された仕様であり、2 つのコンポーネントモデルを統合し、Java を使用した Web ベースのアプリケーション向けプログラミングモデルを大幅に簡略化します。先行する引用符は JSR-299 仕様から除外されました (http://www.jcp.org/en/jsr/detail?id=299 を参照)。

JBoss Enterprise Application Platform には、JSR-299 の参照実装である Weld が含まれます。タイプセーフ依存関係挿入の詳細については、「タイプセーフ依存関係挿入について」を参照してください。

バグを報告する

9.1.3. CDI の利点

- CDI を使用すると、多くのコードをアノテーションに置き換えることにより、コードベースが単純化および削減されます。
- CDI は柔軟であり、CDI を使用すると、挿入およびイベントを無効または有効にしたり、代替の Bean を使用したり、非 CDI オブジェクトを簡単に挿入したりできます。
- CDI で古いコードを使用することは簡単です。これを行うには **beans.xml** を **META-INF/** または **WEB-INF/** ディレクトリーに配置します。このファイルは空白である場合があります。
- CDI を使用すると、パッケージ化とデプロイメントが簡略化され、デプロイメントに追加する必要がある XML の量が減少します。
- CDI により、コンテキストを使用したライフサイクル管理が提供されます。挿入を要求、セッション、会話、またはカスタムコンテキストに割り当てることができます。
- また、CDI により、文字列ベースの挿入よりも安全かつ簡単にデバッグを行える、タイプセーフな依存関係挿入が提供されます。
- CDI はインターセプターと Bean を切り離します。
- CDI では、複雑なイベント通知も提供されます。

バグを報告する

9.1.4. タイプセーフ依存関係挿入について

JSR-299 および CDI 以前は、Java で依存関係を挿入するには文字列を使う方法しかありませんでした。この方法では間違いが起きやすいため、CDI によりタイプセーフな形で依存関係を挿入する機能が導入されました。

CDI の詳細については、「Contexts and Dependency Injection (CDI) について」 を参照してください。

バグを報告する

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

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

*JavaServer Faces (JSF)*は、JSR-314 を実装します。これは、サーバーサイドユーザーインターフェースを構築するための API です。*JBoss Web Framework Kit*には、JavaServer Faces と AJAX の実装である*RichFaces*が含まれます。

*Weld*は、JSR-299 で定義されている*Contexts and Dependency Injection (CDI)*の参照実装です。Weld は、Seam 2 と他の依存関係挿入フレームワークの影響を受けています。Weld は、JBoss Enterprise Application Platform に含まれます。

バグを報告する

9.2. CDI の使用

9.2.1. 最初の手順

9.2.1.1. CDI の有効化

概要

Contexts and Dependency Injection (CDI) は、JBoss Enterprise Application Platform の中核的なテクノロジーの 1 つであり、デフォルトで有効になります。何らかの理由で無効になっている場合は、以下の手順に従って有効にする必要があります。

手順9.1 JBoss Enterprise Application Platform での CDI の有効化

1. **設定ファイルで、CDI サブシステムの詳細がコメントアウトされているかどうかを確認します。**
   サブシステムは、**`domain.xml`** または **`standalone.xml`** 設定ファイルの該当するセクションをコメントアウトするか、該当するセクション全体を削除することにより、無効にできます。

   ***`EAP_HOME`*`/domain/configuration/domain.xml`** または ***`EAP_HOME`*`/standalone/configuration/standalone.xml`** で CDI サブシステムを検索するには、これらのファイルで文字列 **`<extension module="org.jboss.as.weld"/>`** を検索します。検索候補が存在する場合、検索候補は **<extensions>** セクション内部に存在します。

2. **ファイルを編集する前に、JBoss Enterprise Application Platform を停止します。**
   JBoss Enterprise Application Platform により実行中に設定ファイルが変更されるため、設定ファイルを直接編集する前にサーバーを停止する必要があります。

3. **CDI サブシステムを復元するよう設定ファイルを編集します。**
   CDI サブシステムがコメントアウトされている場合は、コメントを削除します。

CDI サブシステムが完全に削除されたら、次の行を、**</extensions>** タグのすぐ上にある新しい行に追加することにより、CDI サブシステムを復元します。

```
<extension module="org.jboss.as.weld"/>
```

4. **JBoss Enterprise Application Platform を再起動します。**
   更新された設定で JBoss Enterprise Application Platform を起動します。

結果

JBoss Enterprise Application Platform は、CDI サブシステムが有効になった状態で起動します。

バグを報告する

9.2.2. CDI を使用したアプリケーションの開発

9.2.2.1. CDI を使用したアプリケーションの開発

はじめに

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

- 「CDI の有効化」
- 「既存のコードでの CDI の使用」
- 「スキャンプロセスからの Bean の除外」
- 「挿入を使用して実装を拡張」
- 「修飾子を使用して不明な挿入を解決」
- 「代替で挿入をオーバーライド」
- 「名前付き Bean の使用」
- 「Bean のライフサイクルの管理」
- 「プロデューサーメソッドの使用」
- 「CDI とのインターセプターの使用」
- 「ステレオタイプの使用」
- 「イベントの発生と確認」

バグを報告する

9.2.2.2. 既存のコードでの CDI の使用

パラメーターがないコンストラクターを持つほぼすべての具象 Java クラスまたはアノテーション @Inject が指定されたコンストラクターは Bean です。Bean の挿入を開始する前に必要な唯一のものは、アーカイブの **META-INF/** または **WEB-INF/** ディレクトリーにある **beans.xml** という名前のファ

イルです。このファイルは空白の場合があります。

手順9.2 CDI アプリケーションでのレガシー Bean の使用

1. **Bean をアーカイブにパッケージ化します。**
   Bean を JAR または WAR アーカイブにパッケージ化します。

2. **`beans.xml` ファイルをアーカイブに含めます。**
   `beans.xml` ファイルを JAR アーカイブの **`META-INF/`** ディレクトリーまたは WAR アーカイブの **`WEB-INF/`** ディレクトリーに配置します。このファイルは空白の場合があります。

結果

これらの Bean を CDI で使用できます。コンテナは、Bean のインスタンスを作成および破棄し、指定されたコンテキストに関連付け、他の Bean に挿入し、EL 式で使用して、修飾子アノテーションで特殊化し、インターセプターとデコレーターをこれらに追加できます (既存のコードを変更しません)。状況によっては、いくつかのアノテーションを追加する必要がある場合があります。

バグを報告する

9.2.2.3. スキャンプロセスからの Bean の除外

概要

Weld の機能の 1 つである JBoss Enterprise Application Platform の CDI 実装は、スキャンからアーカイブのクラスを除外する機能であり、コンテナライフサイクルイベントを発生させ、Bean としてデプロイされます。これは、JSR-299 仕様の一部ではありません。

例9.1 Bean からのパッケージの除外

以下の例では、複数の **<weld:exclude>** タグが使用されています。

1. 最初のタグでは、すべての Swing クラスが除外されます。

2. 2 番目のタグでは、Google Web Toolkit がインストールされていない場合に Google Web Toolkit クラスが除外されます。

3. 3 番目のタグでは、文字列 **`Blether`** (通常の式を使用) で終了するクラスが除外されます (システムプロパティー verbosity が **`low`** に設定されている場合)。

4. 4 番目のタグでは、Java Server Faces (JSF) クラスが除外されます (Wicket クラスが存在し、viewlayer システムプロパティーが設定されてない場合)。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:weld="http://jboss.org/schema/weld/beans"
       xsi:schemaLocation="
          http://java.sun.com/xml/ns/javaee
http://docs.jboss.org/cdi/beans_1_0.xsd
          http://jboss.org/schema/weld/beans
http://jboss.org/schema/weld/beans_1_1.xsd">

    <weld:scan>

        <!-- Don't deploy the classes for the swing app! -->
```

```
        <weld:exclude name="com.acme.swing.**" />

        <!-- Don't include GWT support if GWT is not installed -->
        <weld:exclude name="com.acme.gwt.**">
            <weld:if-class-available name="!com.google.GWT"/>
        </weld:exclude>

        <!--
            Exclude classes which end in Blether if the system property
verbosity is set to low
            i.e.
              java ... -Dverbosity=low
        -->
        <weld:exclude pattern="^(.*)Blether$">
            <weld:if-system-property name="verbosity" value="low"/>
        </weld:exclude>

       <!--
             Don't include JSF support if Wicket classes are present,
and the viewlayer system
             property is not set
        -->
        <weld:exclude name="com.acme.jsf.**">
            <weld:if-class-available name="org.apache.wicket.Wicket"/>
            <weld:if-system-property name="!viewlayer"/>
        </weld:exclude>
    </weld:scan>
</beans>
```

Weld 固有の設定オプションの正式な仕様は http://jboss.org/schema/weld/beans_1_1.xsd で参照できます。

バグを報告する

9.2.2.4. 挿入を使用して実装を拡張

概要

挿入を使用して、既存のコードの機能を追加または変更できます。この例は、既存のクラスに翻訳機能を追加する方法を示しています。翻訳機能は想像上の機能であり、例での実装方法は擬似コードであり、説明を目的としています。

この例では、メソッド **buildPhrase** を持つ Welcome クラスがすでにあることを前提とします。**buildPhrase** メソッドは、都市の名前を引数として取得し、"Welcome to Boston." などのフレーズを出力します。ここでの目標は、このような挨拶を別の言語に翻訳できる **Welcome** クラスのバージョンを作成することです。

例9.2 Translator Bean を Welcome クラスに挿入する

以下の擬似コードは、想像上の **Translator** オブジェクトを **Welcome** クラスに挿入します。**Translator** オブジェクトは、文をある言語から別の言語に翻訳できる EJB ステートレス Bean または別のタイプの Bean になります。この例では、挨拶全体を翻訳するために **Translator**

が使用されます。元の **Welcome** クラスは実際にはまったく変更されません。**Translator** は、**buildPhrase** メソッドが実装される前に挿入されます。

以下のコード例は、サンプル Translating Welcome クラスです。

```
public class TranslatingWelcome extends Welcome {

    @Inject Translator translator;

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

バグを報告する

9.2.3. あいまいな依存関係または満たされていない依存関係

9.2.3.1. 依存性があいまいな場合、あるいは満たされていない場合

コンテナが 1 つの Bean への注入を解決できない場合、依存性があいまいとなります。

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

コンテナは以下の手順を踏み、依存性の解決をはかります。

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

依存性があいまいな場合、あるいは満たされない場合は、コンテナはデプロイメントを中断し例外を送出します。

あいまいな依存性を修正する方法は、「修飾子を使用して不明な挿入を解決」を参照してください。

バグを報告する

9.2.3.2. 修飾子について

修飾子は、Bean を Bean タイプに割り当てるアノテーションです。修飾子を使用すると、挿入する Bean を適切に指定できます。修飾子はリテンションとターゲットを持ちます。これらは、以下の例のように定義されます。

例9.3 @Synchronous 修飾子と @Asynchronous 修飾子を定義する

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

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

例9.4 1@Synchronous 修飾子と @Asynchronous 修飾子を使用する

```
@Synchronous
public class SynchronousPaymentProcessor implements PaymentProcessor {

   public void process(Payment payment) { ... }

}
```

```
@Asynchronous
public class AsynchronousPaymentProcessor implements PaymentProcessor {

   public void process(Payment payment) { ... }
}
```

バグを報告する

9.2.3.3. 修飾子を使用して不明な挿入を解決

概要

このタスクは、不明な挿入を示し、修飾子を使用して不明な挿入を削除します。不明な挿入の詳細については、「依存性があいまいな場合、あるいは満たされていない場合」を参照してください。

例9.5 不明な挿入

Welcome の 2 つの実装があり、1 つは翻訳を行い、もう 1 つは翻訳を行いません。このような場合は、以下の挿入が不明であり、翻訳を行う **Welcome** を使用するよう指定する必要があります。

```
public class Greeter {
  private Welcome welcome;

  @Inject
  void init(Welcome welcome) {
    this.welcome = welcome;
  }
  ...
}
```

手順9.3 修飾子を使用して不明な挿入を解決する

1. **@Translating** という修飾子アノテーションを作成します。

```
@Qualifier
```

```
@Retention(RUNTIME)
@Target({TYPE,METHOD,FIELD,PARAMETERS})
public @interface Translating{}
```

2. **翻訳を行う `Welcome` を `@Translating` アノテーションでアノテートします。**

```
@Translating
public class TranslatingWelcome extends Welcome {
    @Inject Translator translator;
    public String buildPhrase(String city) {
        return translator.translate("Welcome to " + city + "!");
    }
    ...
}
```

3. **挿入の、翻訳を行う `Welcome` を要求します。**
   ファクトリーメソッドパターンの場合と同様に、修飾された実装を明示的に要求する必要があります。不明な点は、挿入時に解決されます。

```
public class Greeter {
  private Welcome welcome;
  @Inject
  void init(@Translating Welcome welcome) {
    this.welcome = welcome;
  }
  public void welcomeVisitors() {
    System.out.println(welcome.buildPhrase("San Francisco"));
  }
}
```

結果

`TranslatingWelcome` が使用されます。不明な点はありません。

バグを報告する

9.2.4. 管理 Bean

9.2.4.1. 管理対象 Beans について

管理 Bean (MBean) は、依存関係の挿入を利用して作成した JavaBean です。各MBean はJava 仮想マシン (JVM) で実行されるリソースを表します。

Java EE 6 はこの定義に基づいて拡張されます。Bean は Java クラスにより実装され、Bean クラスとして参照されます。管理対象 bean は最上位の Java クラスです。

管理対象 Bean の詳細については、JSR-255 仕様 (http://jcp.org/en/jsr/detail?id=255) を参照してください。CDI の詳細については、「Contexts and Dependency Injection (CDI) について」 を参照してください。

バグを報告する

9.2.4.2. Bean であるクラスのタイプ

管理対象 Bean は Java クラスです。管理対象 Bean の基本的なライフサイクルやセマンティクスは、管理対象 Bean の仕様で定義されています。Bean クラス **@ManagedBean** をアノテートすることで明示的に管理対象 Bean を宣言できますが、CDI ではその必要はありません。この仕様によると、CDI コンテナでは、以下の条件を満たすクラスはすべて管理対象 Bean として扱われます。

- 非静的な内部クラスではないこと。
- 具象クラス、あるいは **@Decorator** でアノテートされていること。
- EJB コンポーネントを定義するアノテーションでアノテートされていないこと、あるいは **ejb-jar.xml** で EJB Bean クラスとして宣言されていること。
- インターフェース **javax.enterprise.inject.spi.Extension** が実装されていないこと。
- パラメーターのないコンストラクターか、**@Inject** でアノテートされたコンストラクターがあること。

管理対象 Bean の Bean 型で無制限のものには、直接的あるいは間接的に実装する Bean クラス、全スーパークラス、および全インターフェースが含まれます。

管理対象 Bean にパブリックフィールドがある場合、デフォルトの @Dependent スコープがなければなりません。

バグを報告する

9.2.4.3. CDI を使用してオブジェクトを Bean に挿入する

デプロイメントアーカイブに **META-INF/beans.xml** または **WEB-INF/beans.xml** ファイルが含まれる場合、CDI を使用してデプロイメントの各オブジェクト挿入することが可能です。

この手順では、オブジェクトに他のオブジェクトを挿入する主な方法を紹介します。

1. **@Inject アノテーションを用いてオブジェクトを Bean の一部に挿入します。**
   Bean 内でクラスのインスタンスを取得するには、フィールドに **@Inject** アノテーションを付けます。

   例9.6 TranslateController へ TextTranslator インスタンスを挿入する

   ```
   public class TranslateController {

      @Inject TextTranslator textTranslator;
      ...
   ```

2. **挿入したオブジェクトのメソッドを使用する**
   挿入したオブジェクトのメソッドを直接使用することが可能です。**TextTranslator** にメソッド **translate** があることを前提とします。

   例9.7 挿入したオブジェクトのメソッドを使用する

   ```
   // in TranslateController class

   public void translate() {
   ```

```
    translation = textTranslator.translate(inputText);

}
```

3. **Bean のコンストラクターで挿入を使用する**
   ファクトリーやサービスロケーターを使用して作成する代わりに、Bean のコンストラクターへオブジェクトを挿入することができます。

例9.8 Bean のコンストラクターで挿入を使用する

```
public class TextTranslator {

   private SentenceParser sentenceParser;

   private Translator sentenceTranslator;



   @Inject

   TextTranslator(SentenceParser sentenceParser, Translator
sentenceTranslator) {

      this.sentenceParser = sentenceParser;

      this.sentenceTranslator = sentenceTranslator;

   }

   // Methods of the TextTranslator class
   ...
}
```

4. **Instance(<T>) インターフェースを使用し、プログラムを用いてインスタンスを取得します。**
   Bean 型でパラメーター化されると、**Instance** インターフェースは TextTranslator のインスタンスを返すことができます。

例9.9 プログラムを用いてインスタンスを取得する

```
@Inject Instance<TextTranslator> textTranslatorInstance;

...

public void translate() {

   textTranslatorInstance.get().translate(inputText);

}
```

結果

オブジェクトを Bean に挿入すると、Bean は全オブジェクトのメソッドとプロパティーを使用できるようになります。Bean のコンストラクターに挿入すると、挿入が既に存在するインスタンスを参照する場合以外は、Bean のコンストラクターが呼び出されると挿入されたオブジェクトのインスタンスが作成されます。例えば、セッションのライフタイムの間にセッションスコープ付けされた Bean を挿入しても、新しいインスタンスは作成されません。

バグを報告する

9.2.5. コンテキスト、スコープ、依存関係

9.2.5.1. コンテキストおよびスコープ

CDI の観点から、コンテキストは特定のスコープに関連付けられた Bean のインスタンスを保持するストレージ領域です。

スコープは Bean とコンテキスト間のリンクです。スコープとコンテキストの組み合わせは特定のライフサイクルを持つことがあります。事前定義された複数のスコープが存在し、独自のスコープを作成できます。事前定義されたスコープの例は **`@RequestScoped`**、**`@SessionScoped`**、および **`@ConversationScope`** です。

バグを報告する

9.2.5.2. 利用可能なコンテキスト

表9.1 利用可能なコンテキスト

コンテキスト	説明
@Dependent	Bean は、参照を保持する Bean のライフサイクルにバインドされます。
@ApplicationScoped	アプリケーションのライフサイクルにバインドされます。
@RequestScoped	要求のライフサイクルにバインドされます。
@SessionScoped	セッションのライフサイクルにバインドされます。
@ConversationScoped	会話のライフサイクルにバインドされます。会話スコープは、要求の長さとセッションの間であり、アプリケーションによって制御されます。
カスタムスコープ	上記のコンテキストでニーズが満たされない場合は、カスタムスコープを定義できます。

バグを報告する

9.2.6. Bean ライフサイクル

9.2.6.1. Bean のライフサイクルの管理

概要

このタスクは、要求の残存期間の間 Bean を保存する方法を示しています。他の複数のスコープが存在し、独自のスコープを定義できます。

挿入された Bean のデフォルトのスコープは **@Dependent** です。つまり、Bean のライフサイクルは、参照を保持する Bean のライフサイクルに依存します。詳細については、「コンテキストおよびスコープ」 を参照してください。

手順9.4 Bean ライフサイクルを管理する

1. **必要なスコープに対応するスコープで Bean をアノテートします。**

```
@RequestScoped
@Named("greeter")
public class GreeterBean {
  private Welcome welcome;
  private String city; // getter & setter not shown
  @Inject   void init(Welcome welcome) {
    this.welcome = welcome;
  }
  public void welcomeVisitors() {
    System.out.println(welcome.buildPhrase(city));
  }
}
```

2. **Bean が JSF ビューで使用される場合、Bean はステートを保持します。**

```
<h:form>
  <h:inputText value="#{greeter.city}"/>
  <h:commandButton value="Welcome visitors" action="#
{greeter.welcomeVisitors}"/>
</h:form>
```

結果

Bean は、指定するスコープに関連するコンテキストに保存され、スコープが適用される限り存続します。

- 「Bean プロキシ」
- 「挿入でプロキシを使用する」

バグを報告する

9.2.6.2. プロデューサーメソッドの使用

概要

このタスクは、挿入用の Bean ではないさまざまなオブジェクトを生成するプロデューサーメソッドを使用する方法を示しています。

> **例9.10 代替の代わりにプロデューサーメソッドを使用してデプロイメント後のポリモーフィズムを可能にします。**

例の **@Preferred** アノテーションは、修飾子アノテーションです。修飾子の詳細については、「修飾子について」を参照してください。

```
@SessionScoped
public class Preferences implements Serializable {
   private PaymentStrategyType paymentStrategy;
   ...
   @Produces @Preferred
   public PaymentStrategy getPaymentStrategy() {
       switch (paymentStrategy) {
           case CREDIT_CARD: return new CreditCardPaymentStrategy();
           case CHECK: return new CheckPaymentStrategy();
           default: return null;
       }
   }
}
```

以下の挿入ポイントは、プロデューサーメソッドと同じタイプおよび修飾子アノテーションを持つため、通常の CDI 挿入ルールを使用してプロデューサーメソッドに解決されます。プロデューサーメソッドは、この挿入ポイントを処理するインスタンスを取得するためにコンテナにより呼び出されます。

```
@Inject @Preferred PaymentStrategy paymentStrategy;
```

例9.11 プロデューサーメソッドへのスコープの割り当て

プロデューサーメソッドのデフォルトのスコープは **@Dependent** です。スコープを Bean に割り当てた場合、スコープは適切なコンテキストにバインドされます。この例のプロデューサーメソッドは、1 つのセッションあたり一度だけ呼び出されます。

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy() {
   ...
}
```

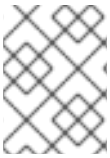
例9.12 プロデューサーメソッド内部での挿入の使用

アプリケーションにより直接インスタンス化されたオブジェクトは、依存関係挿入を利用できず、インターセプターを持ちません。ただし、プロデューサーメソッドへの依存関係挿入を使用して Bean インスタンスを取得できます。

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(CreditCardPaymentStrategy
ccps,
                                          CheckPaymentStrategy cps )
{
   switch (paymentStrategy) {
      case CREDIT_CARD: return ccps;
      case CHEQUE: return cps;
      default: return null;
```

```
    }
}
```

要求スコープ Bean をセッションスコーププロデューサー挿入する場合は、プロデューサーメソッドにより、現在の要求スコープインスタンスがセッションスコープにプロモートされます。これは、適切な動作ではないため、プロデューサーメソッドをこのように使用する場合は注意してください。



注記

プロデューサーメソッドのスコープは、プロデューサーメソッドを宣言する Bean から継承されません。

結果

プロデューサーメソッドを使用して、非 Bean オブジェクトを挿入し、コードを動的に変更できます。

バグを報告する

9.2.7. 名前付き Bean と代替の Bean

9.2.7.1. 名前付き Bean について

Bean には、**@Named** アノテーションを使用して名前が付けられます。Bean を命名することにより、Bean を Java Server Faces (JSF) で直接使用できるようになります。

@Named アノテーションは、Bean 名であるオプションパラメーターを取ります。このパラメーターが省略された場合は、小文字の Bean 名が名前として使用されます。

バグを報告する

9.2.7.2. 名前付き Bean の使用

1. **@Named アノテーションを使用して名前を Bean に割り当てます。**

```
@Named("greeter")
public class GreeterBean {
  private Welcome welcome;

  @Inject
  void init (Welcome welcome) {
    this.welcome = welcome;
  }

  public void welcomeVisitors() {
    System.out.println(welcome.buildPhrase("San Francisco"));
  }
}
```

Bean 名自体はオプションです。省略された場合、クラス名に基づいて Bean に名前が付けられます (最初の文字は小文字になります)。上記の例では、デフォルトの名前は **greeterBean** になります。

2. JSF ビューで名前付き Bean を使用します。

```
<h:form>
  <h:commandButton value="Welcome visitors" action="#
{greeter.welcomeVisitors}"/>
</h:form>
```

結果

名前付き Bean が、JSF ビューでアクションとしてコントロールに割り当てられ、コーディングが最小化されます。

バグを報告する

9.2.7.3. 代替の Bean について

他には、実装が特定のクライアントモジュールまたはデプロイメントシナリオに固有である Bean があります。

例9.13 代替の定義

この代替により、@Synchronous PaymentProcessor と @Asynchronous PaymentProcessor の両方の模擬実装が定義されます。

```
@Alternative @Synchronous @Asynchronous

public class MockPaymentProcessor implements PaymentProcessor {

   public void process(Payment payment) { ... }

}
```

デフォルトでは、@Alternative Bean が無効になります。これらは、**beans.xml** ファイルを編集することにより、特定の Bean アーカイブに対して有効になります。

バグを報告する

9.2.7.4. 代替で挿入をオーバーライド

概要

代替の Bean を使用すると、既存の Bean をオーバーライドできます。これらは、同じ役割を満たすクラスをプラグインする方法として考えることができますが、動作が異なります。これらはデフォルトで無効になります。このタスクは、代替を指定し、有効にする方法を示しています。

手順9.5 挿入をオーバーライドする

このタスクでは、プロジェクトに **TranslatingWelcome** クラスがすでにあることを前提としています。ただし、これを "mock" TranslatingWelcome クラスでオーバーライドするとします。これは、実際の Translator Bean を使用できないテストデプロイメントのケースに該当します。

1. 代替を定義します。

```
@Alternative
```

```
@Translating
public class MockTranslatingWelcome extends Welcome {
  public String buildPhrase(string city) {
    return "Bienvenue Ã  " + city + "!");
  }
}
```

2. **代替を置換します。**
   置換実装をアクティベートするために、完全修飾クラス名を **META-INF/beans.xml** または **WEB-INF/beans.xml**ファイルに追加します。

```
<beans>
  <alternatives>
    <class>com.acme.MockTranslatingWelcome</class>
  </alternatives>
</beans>
```

結果

元の実装の代わりに代替実装が使用されます。

バグを報告する

9.2.8. ステレオタイプ

9.2.8.1. ステレオタイプについて

多くのシステムでは、アーキテクチャーパターンを使用して再帰 Bean ロールのセットを生成します。ステレオタイプを使用すると、このようなロールを指定し、中心的な場所で、このロールを持つ Bean に対する共通メタデータを宣言できます。

ステレオタイプにより、以下のいずれかの組み合わせがカプセル化されます。

- デフォルトスコープ
- インターセプターバインディングのセット

また、ステレオタイプにより、以下の 2 つのいずれかのシナリオを指定できます。

- ステレオタイプを持つすべての Bean にデフォルトの Bean EL 名がある
- ステレオタイプを持つすべての Bean が代替である

Bean では、ステレオタイプをゼロ個、1 個、または複数宣言できます。ステレオタイプアノテーションは、Bean クラスまたはプロデューサーメソッドあるいはフィールドに適用できます。

ステレオタイプは、他の複数のアノテーションをパッケージ化するアノテーションであり、**@Stereotype** でアノテートされます。

ステレオタイプからスコープを継承するクラスは、そのステレオタイプをオーバーライドし、Bean で直接スコープを指定できます。

また、ステレオタイプが **@Named** アノテーションを持つ場合、配置された Bean はデフォルトの Bean 名を持ちます。この Bean は、**@Named** アノテーションが Bean で直接指定された場合に、この名前をオーバーライドできます。名前付き Bean の詳細については、「名前付き Bean について」 を参照してください。

バグを報告する

9.2.8.2. ステレオタイプの使用

概要

ステレオタイプがない場合は、アノテーションをクラスタリングできます。このタスクは、ステレオタイプを使用して煩雑さとコードを減らす方法を示しています。ステレオタイプの詳細については、「ステレオタイプについて」を参照してください。

例9.14 アノテーションの煩雑さ

```
@Secure
@Transactional
@RequestScoped
@Named
public class AccountManager {
  public boolean transfer(Account a, Account b) {
    ...
  }
}
```

手順9.6 ステレオタイプを定義および使用する

1. **ステレオタイプを定義します。**

```
@Secure
@Transactional
@RequestScoped
@Named
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface BusinessComponent {
 ...
}
```

2. **ステレオタイプを使用します。**

```
@BusinessComponent
public class AccountManager {
  public boolean transfer(Account a, Account b) {
    ...
  }
}
```

結果

ステレオタイプにより、コードが削減され、単純化されます。

バグを報告する

9.2.9. オブザーバーメソッド

9.2.9.1. オブサーバーメソッドについて

*オブザーバーメソッド*は、イベント発生時に通知を受け取ります。

CDI は、イベントが発生したトランザクションの*完了前*または*完了後*フェーズ中にイベント通知を受け取る*トランザクションオブザーバーメソッド*を提供します。

バグを報告する

9.2.9.2. イベントの発生と確認

例9.15 イベントの発生

以下のコードは、メソッドで挿入および使用されるイベントを示しています。

```
public class AccountManager {
  @Inject Event<Withdrawal> event;

  public boolean transfer(Account a, Account b) {
    ...
    event.fire(new Withdrawal(a));
  }
}
```

例9.16 修飾子を使用したイベントの発生

修飾子を使用して、より具体的にイベント挿入をアノテートできます。修飾子の詳細については、「修飾子について」を参照してください。

```
public class AccountManager {
  @Inject @Suspicious Event <Withdrawal> event;

  public boolean transfer(Account a, Account b) {
    ...
    event.fire(new Withdrawal(a));
  }
}
```

例9.17 イベントの確認

イベントを確認するには、**@Observes** アノテーションを使用します。

```
public class AccountObserver {
  void checkTran(@Observes Withdrawal w) {
    ...
  }
}
```

例9.18 修飾されたイベントの確認

修飾子を使用して特定の種類のイベントだけを確認できます。修飾子の詳細については、「修飾子について」 を参照してください。

```
public class AccountObserver {
  void checkTran(@Observes @Suspicious Withdrawal w) {
    ...
  }
}
```

バグを報告する

9.2.10. インターセプター

9.2.10.1. インターセプターについて

インターセプターは、JavaBeans 仕様の一部として定義されます (http://jcp.org/aboutJava/communityprocess/final/jsr318/ を参照)。インターセプターを使用すると、Bean のメソッドを直接変更せずに Bean のビジネスメソッドに機能を追加できます。インターセプターは、Bean のビジネスメソッドの前に実行されます。

CDI により、インターセプターと Bean をバインドするアノテーションを利用できるため、この機能が強化されます。

インターセプションポイント

ビジネスメソッドのインターセプション

ビジネスメソッドのインターセプターは、Bean のクライアントによる Bean のメソッド呼び出しに適用されます。

ライフサイクルコールバックのインターセプション

ライフサイクルのコールバックインターセプションは、コンテナによるライフサイクルコールバックの呼び出しに適用されます。

タイムアウトメソッドのインターセプション

タイムアウトメソッドのインターセプターは、コンテナによる EJB タイムアウトメソッドの呼び出しに適用されます。

バグを報告する

9.2.10.2. CDI とのインターセプターの使用

例9.19 CDI なしのインターセプター

CDI がない場合、インターセプターには 2 つの問題があります。

- Bean は、インターセプター実装を直接指定する必要があります。
- アプリケーションの各 Bean は、インターセプターの完全なセットを適切な順序で指定する必要があります。この場合、アプリケーション全体でインターセプターを追加または削除するには時間がかかり、エラーが発生する傾向があります。

```
@Interceptors({
  SecurityInterceptor.class,
  TransactionInterceptor.class,
  LoggingInterceptor.class
})
@Stateful public class BusinessComponent {
  ...
}
```

手順9.7 CDI とのインターセプターの使用

1. インターセプターバインディングタイプを定義します。

```
@InterceptorBinding
@Retention(RUNTIME)
@Target({TYPE, METHOD})
public @interface Secure {}
```

2. インターセプター実装をマークします。

```
@Secure
@Interceptor
public class SecurityInterceptor {
  @AroundInvoke
  public Object aroundInvoke(InvocationContext ctx) throws Exception
{
    // enforce security ...
    return ctx.proceed();
    }
}
```

3. ビジネスコードでインターセプターを使用します。

```
@Secure
public class AccountManager {
  public boolean transfer(Account a, Account b) {
    ...
  }
}
```

4. インターセプターを **META-INF/beans.xml** または **WEB-INF/beans.xml** に追加することにより、インターセプターをデプロイメントで有効にします。

```
<beans>
  <interceptors>
    <class>com.acme.SecurityInterceptor</class>
    <class>com.acme.TransactionInterceptor</class>
  </interceptors>
</beans>
```

インターセプターは、リストされた順序で適用されます。

結果

CDI により、インターセプターコードが単純化され、ビジネスコードへの適用が簡単になります。

バグを報告する

9.2.11. デコレーターについて

デコレーターは、特定の Java インターフェースからの呼び出しをインターセプトし、そのインターフェースに割り当てられたすべてのセマンティクスを認識します。デコレーターは、何らかの業務をモデル化するのに役に立ちますが、インターセプターの一般性を持ちません。デコレーターは Bean または抽象クラスであり、デコレートするタイプを実装し、**@Decorator** でアノテートされます。

例9.20 デコレーターの例

```
@Decorator
public abstract class LargeTransactionDecorator
      implements Account {
   @Inject @Delegate @Any Account account;
   @PersistenceContext EntityManager em;
   public void withdraw(BigDecimal amount) {
      ...
   }
   public void deposit(BigDecimal amount);
      ...
   }
}
```

バグを報告する

9.2.12. 移植可能な拡張機能について

CDI は、フレームワーク、拡張機能、および他のテクノロジーとの統合の基礎となることを目的としています。したがって、CDI は、移植可能な CDI の拡張機能の開発者が使用する SPI のセットを公開します。拡張機能は、以下の種類の機能を提供できます。

- ビジネスプロセス管理エンジンとの統合
- Spring、Seam、GWT、Wicket などのサードパーティーフレームワークとの統合
- CDI プログラミングモデルに基づく新しいテクノロジー

JSR-299 仕様に基づいて、移植可能な拡張機能は次の方法でコンテナと統合できます。

- 独自の Bean、インターセプター、およびデコレーターをコンテナに提供します。
- 依存関係挿入サービスを使用して独自のオブジェクトに依存関係を挿入します。
- カスタムスコープのコンテキスト実装を提供します。
- アノテーションベースのメタデータを他のソースからのメタデータで拡大またはオーバーライドします。

バグを報告する

9.2.13. Bean プロキシ

9.2.13.1. Bean プロキシ

プロキシは Bean のサブクラスでランタイム時に生成されます。プロキシは Bean の作成時に挿入され、依存 Bean のライフサイクルはプロキシに関係しているため、依存するスコープ付き Bean をプロキシから挿入することができます。また、プロキシは依存関係の挿入の代わりとして使用され、2 つの問題を解決します。

プロキシを利用することで解決される依存関係挿入の問題

- パフォーマンス - プロキシは依存関係の挿入よりも速度が早いため、高パフォーマンスを必要とする Bean に利用することができます。
- スレッドセーフ - 複数のスレッドが同時に Bean にアクセスしている場合でも、プロキシは適切な Bean インスタンスにリクエストを転送します。依存関係の挿入はスレッドの安全性を保証しません。

プロキシ化できないクラス型

- プリミティブ型あるいはアレイ型
- `final` のクラスあるいは `final` メソッドを持つクラス
- プライベートではないデフォルトのコンストラクターを持つクラス

バグを報告する

9.2.13.2. 挿入でプロキシを使用する

概要

各 Bean のライフサイクルが異なる場合に挿入にプロキシが使用されます。プロキシはランタイム時に作成された Bean のサブクラスで、Bean クラスのプライベートメソッド以外のメソッドをすべて上書きします。プロキシは実際の Bean インスタンスへ呼び出しを転送します。

この例では、`PaymentProcessor` インスタンスは直接 `Shop` へ挿入されません。その代わりにプロキシが挿入され、`processPayment()` メソッドが呼び出されるとプロキシが現在の `PaymentProcessor` Bean インスタンスをルックアップし、`processPayment()` メソッドを呼び出します。

> **例9.21 プロキシの挿入**

```
@ConversationScoped
class PaymentProcessor
{
  public void processPayment(int amount)
  {
    System.out.println("I'm taking $" + amount);
  }
}

@ApplicationScoped
public class Shop
{

  @Inject
  PaymentProcessor paymentProcessor;

  public void buyStuff()
  {
    paymentProcessor.processPayment(100);
  }
}
```

プロキシ化できるクラス型などプロキシに関する詳細情報は 「Bean プロキシ」 を参照してください。

バグを報告する

第10章 JAVA トランザクション API (JTA)

10.1. 概要

10.1.1. Java トランザクション API (JTA) の概要

はじめに

これらのトピックは、 Java トランザクション API (JTA) の基礎的な内容について取り上げます。

- 「Java Transactions API (JTA) について」
- 「JTA トランザクションのライフサイクル」
- 「JTA トランザクションの例」

バグを報告する

10.2. トランザクションの概念

10.2.1. トランザクションについて

トランザクションは 2 つ以上のアクションで構成されており、アクションすべてが成功または失敗する必要があります。成功した場合はコミット、失敗した場合はロールバックが結果的に実行されます。ロールバックでは、トランザクションがコミットを試行する前に、各メンバーのステートが元の状態に戻ります。

適切に設計されたトランザクションは一般的に *ACID* (原子性、一貫性、独立性、永続性) を基準とします。

バグを報告する

10.2.2. トランザクションの ACID プロパティーについて

ACID は 原子性 (**Atomicity**)、一貫性 (**Consistency**)、独立性 (**Isolation**)、永続性 (**Durability**) の略語です。この単語は通常データベースやトランザクション操作において使用されます。

ACID の定義

原子性 (Atomicity)

トランザクションの原子性を保つには、トランザクション内の全メンバーが同じ決定をする必要があります。つまり、全メンバーがコミットまたはロールバックを行う必要があります。原子性が保たれない場合の結果を *ヒューリスティックな結果* と言います。

一貫性 (Consistency)

一貫性とは、データベーススキーマの観点から、データベースに書き込まれたデータが有効なデータであることを保証するという意味です。データベースあるいは他のデータソースは常に一貫した状態でなければなりません。一貫性のない状態の例には、操作が中断される前にデータの半分が書き込みされてしまったフィールドなどがあります。すべてのデータが書き込まれた場合や、書き込みが完了しなかった時に書き込みがロールバックされた場合に、一貫した状態となります。

独立性 (Isolation)

独立性とは、トランザクションのスコープ外のプロセスがデータを変更できないように、トランザクションで操作されたデータが変更前にロックされる必要があることを意味します。

永続性 (Durability)

永続性とは、トランザクションのメンバーにコミットの指示を出してから外部で問題が発生した場合、問題が解決されると全メンバーがトランザクションのコミットを継続できるという意味です。ここで言う問題とは、ハードウェア、ソフトウェア、ネットワークなどのシステムが関係する問題のことです。

バグを報告する

10.2.3. トラザクションコーディネーターあるいはトランザクションマネージャーについて

JBoss Enterprise Application Platform のトランザクションでは、*トランザクションコーディネーター*と*トランザクションマネージャー*の用語はほとんど同じことを意味します。トランザクションコーディネーターという用語は通常、分散トランザクションで利用されます。

JTA トランザクションでは、*トランザクションマネージャー*は JBoss Enterprise Application Platform 内で実行され、*2 相コミット*のプロトコルでトランザクションの参加者と通信します。

トランザクションマネージャーはトランザクションの参加者に対して、別のトランザクションの参加者の結果に従い、データをコミットするか、ロールバックするか指示を出します。こうすることで、確実にトランザクションが ACID 基準に準拠するようにします。

JTS トランザクションでは、トランザクションコーディネーターは別サーバーにある各種トランザクションマネージャー同士のやり取りを管理します。

- 「トランザクションの参加者を表示します。」
- 「トランザクションの ACID プロパティーについて」
- 「2 相コミットプロトコルについて」

バグを報告する

10.2.4. トランザクションの参加者を表示します。

トランザクションの参加者とはトランザクション内のプロセスのことで、ステートをコミットあるいはロールバックする機能を持ちます。データベースや他のアプリケーションなどがその一例です。トランザクションの各参加者は、他のすべての参加者がステートのコミットに合意した場合にのみステートをコミットします。その他の場合は、各参加者はロールバックを実行します。

- 「トランザクションについて」
- 「トラザクションコーディネーターあるいはトランザクションマネージャーについて」

バグを報告する

10.2.5. Java Transactions API (JTA) について

Java Transactions API (JTA) は、Java Enterprise Edition アプリケーションでトランザクションを利用する際の仕様で、JSR-907 に定義されています。

JTA トランザクションは複数のアプリケーションサーバーにまたがって分散されず、ネストすることはできません。

JTA トランザクションは EJB コンテナーで制御されます。アノテーションは、お使いのコード内でトランザクションを作成および制御する 1 つの方法です。

バグを報告する

10.2.6. Java Transaction Service (JTS) について

*Java トランザクションサービス (JTS)*は、トランザクションの参加者が複数の Java Enterprise Edition コンテナ (アプリケーションサーバー) に存在する場合に Java Transaction API (JTA) トランザクションをサポートするメカニズムです。ローカル JTA トランザクションの場合と同様に、各コンテナは*トランザクションマネージャー (TM)* と呼ばれるプロセスを実行します。TM は、*Common Object Request Broker Architecture (CORBA)* と呼ばれる通信標準を使用して *Object Request Broker (ORB)* と呼ばれるプロセスでお互いと通信します。

アプリケーションの観点から、JTS トランザクションは JTA トランザクションと同様に動作します。違いは、トランザクションの参加者とデータソースが別のコンテナに存在することです。

注記

JBoss Enterprise Application Platform に含まれる JTS の実装は、*分散 JTA トランザクション*をサポートします。分散 JTA トランザクションと完全準拠 JTS トランザクションの違いは外部のサードパーティー ORB との相互運用性です。この機能は、JBoss Enterprise Application Platform 6 ではサポートされません。サポートされる設定では、複数の JBoss Enterprise Application Platform コンテナでのみトランザクションが分散されます。

- 「分散トランザクションについて」
- 「トラザクションコーディネーターあるいはトランザクションマネージャーについて」

バグを報告する

10.2.7. XA データソースおよび XA トランザクションについて

XA データソースとは XA のグローバルトランザクションに参加できるデータソースのことです。

XA トランザクションとは、複数のリソースにまたがることができるトランザクションのことです。これには、コーディネートを行うトランザクションマネージャーが関係します。このトランザクションマネージャーは、すべてが 1 つのグローバルトランザクションに関与する 1 つ以上のデータベースまたはその他のトランザクションリソースを持ちます。

バグを報告する

10.2.8. XA リカバリーについて

Java トランザクション API (JTA) は複数の *X/Open XA リソース*にまたがる分散トランザクションを許可します。XA は *Extended Architecture* (拡張アーキテクチャー) の略で、複数のバックエンドデータストアを使用するトランザクションを定義するため X/Open Group によって開発されました。XA 標準には、グローバル *トランザクションマネージャー (TM)* とローカルリソースマネージャーとの間のインターフェースに関する説明があります。XA は、トランザクションの原子性を保持しながらアプリケーションサーバー、データベース、キャッシュ、メッセージキューなどの複数のリソースが同じトラン

ザクションに参加できるようにします。原子性とは、参加者の 1 つが変更のコミットに失敗した場合に他の参加者がトランザクションをアボートし、トランザクションが発生する前の状態に戻すことを言います。

*XA リカバリー*は、トランザクションの参加者がクラッシュしたり使用できなくなったりしても、トランザクションの影響を受けたすべてのリソースが確実に更新またはロールバックされるようにするプロセスのことです。JBoss Enterprise Application Platform 6 の範囲内では、XA データソースや JMS メッセージキュー、JCA リソースアダプターなどの XA リソースやサブシステムに対して、トランザクションサブシステムが XA リカバリーのメカニズムを提供します。

XA リカバリーはユーザーの介入がなくても発生します。XA リカバリーに失敗すると、エラーがログ出力に記録されます。サポートが必要な場合は Red Hat グローバルサポートサービスまでご連絡ください。

バグを報告する

10.2.9. 2 相コミットプロトコルについて

2 相コミット (2PC) とはデータベーストランザクションでの通常のパターンのことです。

フェーズ 1

最初のフェーズでは、トランザクションをコミットできるか、あるいはロールバックする必要があるかをトランザクションの参加者がトランザクションコーディネーターに通知します。

フェーズ 2

2 番目のフェーズでは、トランザクションコーディネーターが全体のトランザクションをコミットするか、ロールバックするか決定します。参加者が 1 つでもコミットできない場合、トランザクションはロールバックしなければなりません。参加者がすべてコミットできる場合はトランザクションはコミットすることができます。コーディネーターは何を行うかをトランザクションに指示し、トランザクションは何を行ったかコーディネーターに通知します。この時点で、トランザクションが完了します。

バグを報告する

10.2.10. トランザクションタイムアウトについて

原子性を確保し、トランザクションを ACID 標準に準拠させるため、トランザクションの一部が長期間実行される場合があります。トランザクションの参加者は、コミット時にデータソースの一部をロックする必要があります。また、トランザクションマネージャーは各トランザクション参加者からの応答を待ってからすべての参加者にコミットあるいはロールバックの指示を出す必要があります。ハードウェアあるいはネットワークの障害のため、リソースが永久にロックされることがあります。

トランザクションのタイムアウトをトランザクションと関連付け、ライフサイクルを制御することができます。タイムアウトのしきい値がトランザクションのコミットあるいはロールバック前に渡された場合、タイムアウトにより、自動的にトランザクションがロールバックされます。

トランザクションサブシステム全体に対しデフォルトのタイムアウト値を設定できます。または、デフォルトのタイムアウト値を無効にし、トランザクションごとにタイムアウトを指定できます。

バグを報告する

10.2.11. 分散トランザクションについて

*分散トランザクション*あるいは*分散 Java Transaction API (JTA) トランザクション*は、複数の Enterprise Application Platform サーバー上に参加者が存在するトランザクションです。 分散トランザクションと *Java Transaction Service (JTS) トランザクション*との違いは、JTS の仕様では異なるベン

ダーのアプリケーションサーバーにまたがってトランザクションが分散可能でなければならない点です。JBoss Enterprise Application Platform は分散 JTA トランザクションに対応しています。

バグを報告する

10.2.12. ORB 移植性 API について

Object Request Broker (ORB) とは、複数のアプリケーションサーバーにわたって分散されるトランザクションの参加者、コーディネーター、リソース、その他のサービスにメッセージを送受信するプロセスのことです。ORB は標準的なインターフェース記述言語 (IDL) を使用してメッセージを通信し解釈します。*Common Object Request Broker Architecture (CORBA)*は JBoss Enterprise Application Platform の ORB によって使用される IDL です。

ORB を使用する主なタイプのサービスは、Java トランザクションサービス (JTS) プロトコルを使用する分散 Java トランザクションのシステムです。レガシーシステムなどの他のシステムは、通信にリモートエンタープライズ JavaBeans や JAX-WS または JAX-RS Web サービスなどのメカニズムを使用せずに、 ORB を使用することがあります。

ORB 移植性 API は ORB とやりとりするメカニズムを提供します。この API は ORB への参照を取得するメソッドや、ORB からの受信接続をリッスンするモードにアプリケーションを置くメソッドを提供します。API のメソッドの一部はすべての ORB によってサポートされていません。このような場合、例外がスローされます。

API は 2 つの異なるクラスによって構成されます。

ORB 移植性 API のクラス

- `com.arjuna.orbportability.orb`
- `com.arjuna.orbportability.oa`

ORB 移植性 API に含まれるメソッドやプロパティーの詳細は、Red Hat カスタマーポータルで JBoss Enterprise Application Platform 6 の Javadocs バンドルを参照してください。

バグを報告する

10.2.13. ネストされたトランザクションについて

ネストされたトランザクションは、一部の参加者がトランザクションでもあるトランザクションです。

ネストされたトランザクションの利点

障害の分離

サブトランザクションがロールバックされた場合に、使用しているオブジェクトが失敗したため、エンクローズトランザクションはロールバックする必要がありません。

モジュール性

新しいトランザクションが開始されるときにトランザクションがすでに呼び出しに関連付けられている場合は、新しいトランザクションがそのトランザクション内にネストされます。したがって、オブジェクトでトランザクションが必要なことがわかっている場合は、オブジェクト内でトランザクションを開始できます。オブジェクトのメソッドがクライアントトランザクションなしで呼び出された場合は、オブジェクトのトランザクションは最上位レベルです。それ以外の場合、これらの

トランザクションはクライアントのトランザクションのスコープ内でネストされます。同様に、クライアントはオブジェクトがトランザクション対応であるかどうかを知る必要がありません。クライアントは、独自のトランザクションを開始できます。

ネストされたトランザクションは、Java トランザクション API (JTA) の一部ではなく Java トランザクションサービス (JTS) API の一部としてのみサポートされます。(非分散) JTA トランザクションをネストしようとすると、例外が発生します。

バグを報告する

10.3. トランザクションの最適化

10.3.1. トランザクション最適化の概要

はじめに

JBoss Enterprise Application Platform のトランザクションサブシステムには複数の最適化機能が含まれており、お使いのアプリケーションでご活用いただけます。

- 「推定中止 (presumed-abort) 最適化について」
- 「読み取り専用の最適化について」
- 「1 相コミット (1PC) の LRCO 最適化について」

バグを報告する

10.3.2. 1 相コミット (1PC) の LRCO 最適化について

トランザクションでは、一般的に 2 相コミットプロトコル (2PC) が使用されることが多いですが、両フェーズに対応する必要がなかったり、対応できない場合もあります。そのような場合、*1 相コミット (1PC)* プロトコルを使用することができます。XA 未対応のデータソースがトランザクションに参加する必要がある場合などがこの一例になります。

このような状況では、*最終リソースコミット最適化 (LRCO: Last Resource Commit Optimization)*という最適化が適用されます。1 相リソースは、トランザクションの準備フェーズで最後に処理され、コミットが試行されます。コミットに成功すると、トランザクションログが書き込まれ、残りのリソースが 2PC に移動します。最終リソースがコミットに失敗すると、トランザクションはロールバックされます。

このプロトコルにより、トランザクションの多くが通常に完了できますが、特殊なエラーによりトランザクションの結果に一貫性がなくなってしまう場合もあります。そのため、この方法は最終手段としてお使いください。

ローカルの TX データソースが 1 つのみトランザクションで使用されると、LRCO が自動的に適用されます。

- 「2 相コミットプロトコルについて」

バグを報告する

10.3.3. 推定中止 (presumed-abort) 最適化について

トランザクションをロールバックする場合、この情報をローカルで記録し、参加している参加者すべてに通知します。この通知は形式的なもので、トランザクションの結果には何ら影響を与えません。全参加者に通知がいくと、このトランザクションに関する情報を削除することができます。

トランザクションのステートに対する後続の要求が発生すると、利用可能な情報がなくなります。このような場合、要求側はトランザクションが中断され、ロールバックされたと推測します。*推定中止 (presumed-abort)* の最適化とは、トランザクションがコミットの実行を決定するまでは参加者に関する情報を永続化する必要がないことを意味します。これは、トランザクションがコミットの実行を決定する時点以前に発生した障害はトランザクションの中止であると推定されるためです。

バグを報告する

10.3.4. 読み取り専用の最適化について

参加者が準備するよう要求されると、トランザクション中に変更したデータがないことをコーディネーターに伝えることができます。参加者が最終的にどうなってもトランザクションに影響を与えることはないため、このような参加者にトランザクションの結果について通知する必要はありません。*読み取り専用*の参加者はコミットプロトコルの第二フェーズから省略可能です。

バグを報告する

10.4. トランザクションの結果

10.4.1. トランザクションの結果について

可能なトランザクションの結果は次の 3 つになります。

ロールバック

トランザクションの参加者のいずれかがコミットできなかったり、トランザクションコーディネーターが参加者にコミットを指示できない場合、トランザクションがロールバックされます。詳細は「トランザクションロールバックについて」を参照してください。

コミット

トランザクションの参加者すべてがコミットできる場合、トランザクションコーディネーターはコミットの実行を指示します。詳細は「トランザクションのコミットについて」を参照してください。

ヒューリスティックな結果

トランザクションの参加者の一部がコミットし、他の参加者がロールバックした場合をヒューリスティックな結果と呼びます。ヒューリスティックな結果が発生すると、人的な介入が必要になります。詳細は「ヒューリスティックな結果について」を参照してください。

バグを報告する

10.4.2. トランザクションのコミットについて

トランザクションの参加者がコミットすると、新規のステートが永続化されます。新規のステートはトランザクションで作業を行った参加者により作成されます。トランザクションのメンバーがデータベースに記録を書き込む時などが最も一般的な例になります。

コミット後、トランザクションの情報はトランザクションコーディネーターから削除され、新たに書き込まれたステートが永続ステートとなります。

バグを報告する

10.4.3. トランザクションロールバックについて

トランザクションの参加者はトランザクションの開始前に、ステートを反映するためステートをリストアし、ロールバックを行います。ロールバック後のステートはトランザクション開始前のステートと同じになります。

バグを報告する

10.4.4. ヒューリスティックな結果について

ヒューリスティックな結果あるいは原子的でない結果とは、トランザクションに異常があることで、トランザクションの参加者の一部はステートをコミットし、その他の参加者はロールバックした場合を言います。ヒューリスティックな結果が発生すると、ステートの一貫性が保たれなくなります。

通常、ヒューリスティックな結果は、2 相コミット (2PC) プロトコルの 2 番目のフェーズで発生します。基盤のハードウェアや基盤サーバーの通信サブシステムの障害が原因となる場合が多くあります。

ヒューリスティックな結果には 4 種類あります。

ヒューリスティックロールバック

参加者の一部あるいはすべてが一方的にトランザクションをロールバックしたため、コミット操作に失敗します。

ヒューリスティックコミット

参加者のすべてが一方的にコミットしたため、ロールバック操作に失敗します。例えば、コーディネーターが正常にトランザクションを準備したにも関わらず、ログ更新の失敗などコーディネーター側で障害が発生したためロールバックの実行を決定した場合などに発生します。暫定的に参加者がコミットの実行を決定する場合があります。

ヒューリスティック混合

一部の参加者がコミットし、その他の参加者はロールバックした状態です。

ヒューリスティックハザード

更新の一部の結果が不明な状態です。既知の更新結果はすべてコミットまたはロールバックします。

ヒューリスティックな結果が起こると、システムの整合性が保たれなくなり、通常、解決に人的介入が必要になります。ヒューリスティックな結果に依存するようなコードは記述しないようにしてください。

- 「2 相コミットプロトコルについて」

バグを報告する

10.4.5. JBoss Transactions エラーと例外

`UserTransaction` クラスのメソッドがスローする例外に関する詳細は、http://download.oracle.com/javaee/1.3/api/javax/transaction/UserTransaction.html の『UserTransaction API』 の仕様を参照してください。

バグを報告する

10.5. JTA トランザクションの概要

10.5.1. Java Transactions API (JTA) について

Java Transactions API (JTA) は、Java Enterprise Edition アプリケーションでトランザクションを利用する際の仕様で、JSR-907 に定義されています。

JTA トランザクションは複数のアプリケーションサーバーにまたがって分散されず、ネストすることはできません。

JTA トランザクションは EJB コンテナーで制御されます。アノテーションは、お使いのコード内でトランザクションを作成および制御する 1 つの方法です。

バグを報告する

10.5.2. JTA トランザクションのライフサイクル

リソースがトランザクションへの参加を要求すると、一連のイベントが開始されます。*トランザクションマネージャー*は、アプリケーションサーバー内のプロセスで、トランザクションを管理します。*トランザクションの参加者*は、トランザクションに参加するオブジェクトです。また、*リソース*とは、データソースや JMS 接続ファクトリ、その他の JCA 接続のことです。

1. **アプリケーションが新しいトランザクションを開始する**
   トランザクションを開始するには、お使いのアプリケーションが JNDI から(または、EJB の場合はアノテーションから) **`UserTransaction`** クラスのインスタンスを取得します。**`UserTransaction`** インターフェースには、トップレベルのトランザクションを開始、コミット、ロールバックするメソッドが含まれています。新規作成されたトランザクションは、そのトランザクションを呼び出すスレッドと自動的に関連付けされます。ネストされたトランザクションは JTA ではサポートされないため、すべてのトランザクションがトップレベルのトランザクションとなります。

   `UserTransaction.begin()` を呼び出すと、新規トランザクションが開始されます。この時点移行に使ったリソースはすべて、このトランザクションと関連付けられます。1つ以上のリソースが登録された場合、このトランザクションは XA トランザクションになり、コミット時に 2 相コミットプロトコルに参加します。

2. **アプリケーションがステートを変更する**
   次に、トランザクションが作業を実行しステートを変更します。

3. **アプリケーションがコミットまたはロールバックを決定する**
   お使いのアプリケーションがステータスの変更を終了すると、コミットするか、ロールバックするか決定し、適切なメソッドを呼び出します。**`UserTransaction.commit()`** または **`UserTransaction.rollback()`** を呼び出します。1 つ以上のリソースを登録している場合は、ここで 2 相コミットプロトコル (2PC) が起こります。「2 相コミットプロトコルについて」

4. **トランザクションマネージャーが記録からトランザクションを削除する**
   コミットあるいはロールバックが完了すると、トランザクションマネージャーは記録を消去し、トランザクションに関する情報を削除します。

障害回復

障害回復は自動的に行われます。リソース、トランザクションの参加者、アプリケーションサーバーが使用できなくなった場合、この問題が解決した時にトランザクションマネージャーがリカバリー処理を行います。

- 「トランザクションについて」
- 「トラザクションコーディネーターあるいはトランザクションマネージャーについて」
- 「 トランザクションの参加者を表示します。」
- 「2 相コミットプロトコルについて」
- 「XA データソースおよび XA トランザクションについて」

バグを報告する

10.6. トランザクションサブシステムの設定

10.6.1. トランザクション設定の概要

はじめに

次の手順は、JBoss Enterprise Application Platform のトランザクションサブシステムを設定する方法を示しています。

- 「JTA トランザクションを使用するようにデータソースを設定」
- 「XA Datasource の設定」
- 「トランザクションマネージャーの設定」
- 「トランザクションサブシステムのログ設定」

バグを報告する

10.6.2. トランザクションデータソースの設定

10.6.2.1. JTA トランザクションを使用するようにデータソースを設定

概要

ここでは、データソースで Java Transactions API (JTA) を有効にする方法を説明します。

要件

このタスクを行う前に、 次の条件を満たす必要があります。

- お使いのデータベースまたはその他のリソースが JTA をサポートしている必要があります。不明な場合は、データソースまたはリソースの文書を参照してください。
- データベースを作成する必要があります。「管理インターフェースによる非 XA データソースの作成」 を参照してください。
- JBoss Enterprise Application Platform を停止します。
- テキストエディターで設定ファイルを直接編集できる権限を持たなければなりません。

手順10.1 JTA トランザクションを使用するようデータソースを設定する

1. **テキストエディターで設定ファイルを開きます。**
   JBoss Enterprise Application Platform を管理対象ドメインまたはスタンドアロンサーバーで実行するかによって、設定ファイルの場所は異なります。

   - **管理対象ドメイン**
     管理対象ドメインのデフォルトの設定ファイルは、Red Hat Enterprise Linux の場合は ***EAP_HOME*/domain/configuration/domain.xml** にあります。Microsoft Windows サーバーの場合は ***EAP_HOME*\domain\configuration\domain.xml** にあります。

   - **スタンドアロンサーバー**
     スタンドアロンサーバーのデフォルトの設定ファイルは、Red Hat Enterprise Linux の場合は ***EAP_HOME*/standalone/configuration/standalone.xml** にあります。Microsoft Windows サーバーの場合は ***EAP_HOME*\standalone\configuration\standalone.xml** にあります。

2. **お使いのデータソースに対応する<datasource> タグを探します。**
   データソースの **jndi-name** 属性には作成時に指定した属性が設定されます。例えば、ExampleDS データソースは次のようになります。

   ```
   <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-
   name="H2DS" enabled="true" jta="true" use-java-context="true" use-
   ccm="true">
   ```

3. **jta 属性を true に設定します。**
   上記のように、**jta="true"** を **<datasource>** タグの内容に追加します。

4. **設定ファイルを保存します。**
   設定ファイルを保存しテキストエディターを終了します。

5. **JBoss Enterprise Application Platform を起動します。**
   JBoss Enterprise Application Platform 6 サーバーを再起動します。

結果

JBoss Enterprise Application Platform が起動し、データソースが JTA トランザクションを使用するように設定されます。

バグを報告する

10.6.2.2. XA Datasource の設定

要件

XA Datasource を追加するには、管理コンソールにログインする必要があります。詳細は「管理コンソールへログイン」 を参照してください。

1. **新しいデータソースの追加**
   新しいデータソースを JBoss Enterprise Application Platform に追加します。「管理インターフェースによる非 XA データソースの作成」 の手順に従いますが、上部の **XA Datasource** タブをクリックしてください。

2. **必要に応じて他のプロパティーを設定します。**
   データソースパラメーターの一覧は「データソースのパラメーター」にあります。

結果

XA Datasource が設定され、使用する準備ができます。

バグを報告する

10.6.2.3. 管理コンソールへログイン

要件

- JBoss Enterprise Application Platform 6 が稼働している必要があります。

手順10.2 管理コンソールへログイン

1. **管理コンソールのスタートページに移動**
   Web ブラウザーで管理コンソールに移動します。デフォルトの場所は http://localhost:9990/console/ です。ポート 9990 は管理コンソールのソケットバインディングとして事前定義されています。

2. **管理コンソールへログイン**
   以前作成したアカウントのユーザー名とパスワードを入力し、管理コンソールのログイン画面でログインします。

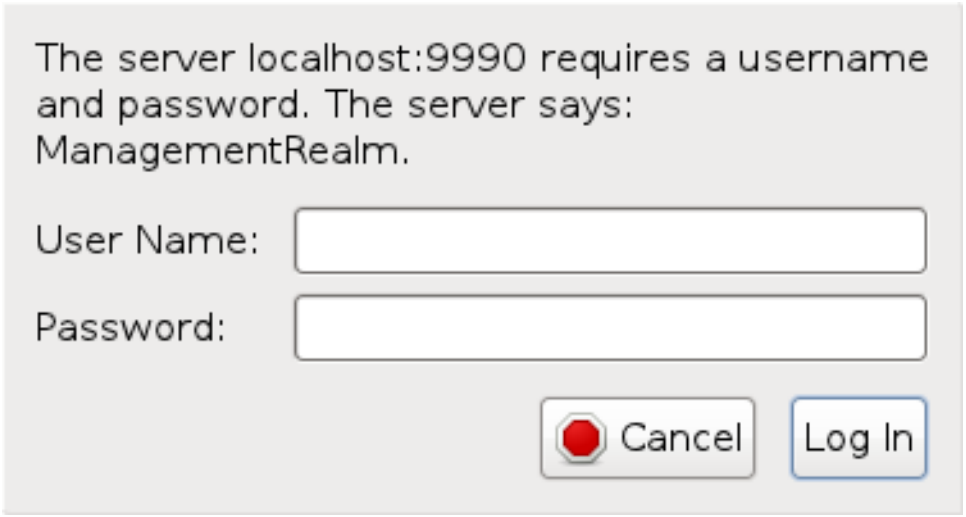


図10.1 管理コンソールのログイン画面

結果

ログインすると、管理コンソールの最初のページが表示されます。

管理対象ドメイン

http://localhost:9990/console/App.html#server-instances

スタンドアロンサーバー

http://localhost:9990/console/App.html#server-overview

バグを報告する

10.6.2.4. 管理インターフェースによる非 XA データソースの作成

概要

ここでは、管理コンソールまたは管理 CLI のいずれかを使用して非 XA データソースを作成する手順について取り上げます。

要件

- JBoss Enterprise Application Platform 6 サーバーが稼働している必要があります。

注記

バージョン 10.2 以前の Oracle データソースでは非トランザクション接続とトランザクション接続が混在するとエラーが発生したため、**<no-tx-separate-pools/>** パラメーターが必要でした。一部のアプリケーションでは、このパラメーターが必要ではなくなりました。

手順10.3 管理 CLI または管理コンソールのいずれかを使用したデータソースの作成

- **管理 CLI**
  a. CLI ツールを起動し、サーバーに接続します。
  b. 以下のコマンドを実行して非 XA データソースを作成し、適切に変数を設定します。

```
data-source add --name=DATASOURCE_NAME --jndi-name=JNDI_NAME --driver-name=DRIVER_NAME  --connection-url=CONNECTION_URL
```

  c. データソースを有効にします。

```
data-source enable --name=DATASOURCE_NAME
```

- **管理コンソール**
  a. 管理コンソールへログインします。
  b. **管理コンソールの [Datasources] パネルに移動します。**
    i. ■ **スタンドアロンモード**
    コンソールの右上より **[Profile]** タブを選択します。
    ■ **ドメインモード**
    A. コンソールの右上より **[Profiles]** タブを選択します。
    B. 左上のドロップダウンボックスより該当するプロファイルを選択します。
    C. コンソールの左側にある **[Subsystems]** メニューを展開します。
    ii. コンソールの左側にあるメニューより **[Connector]** → **[Datasources]** と選択します。

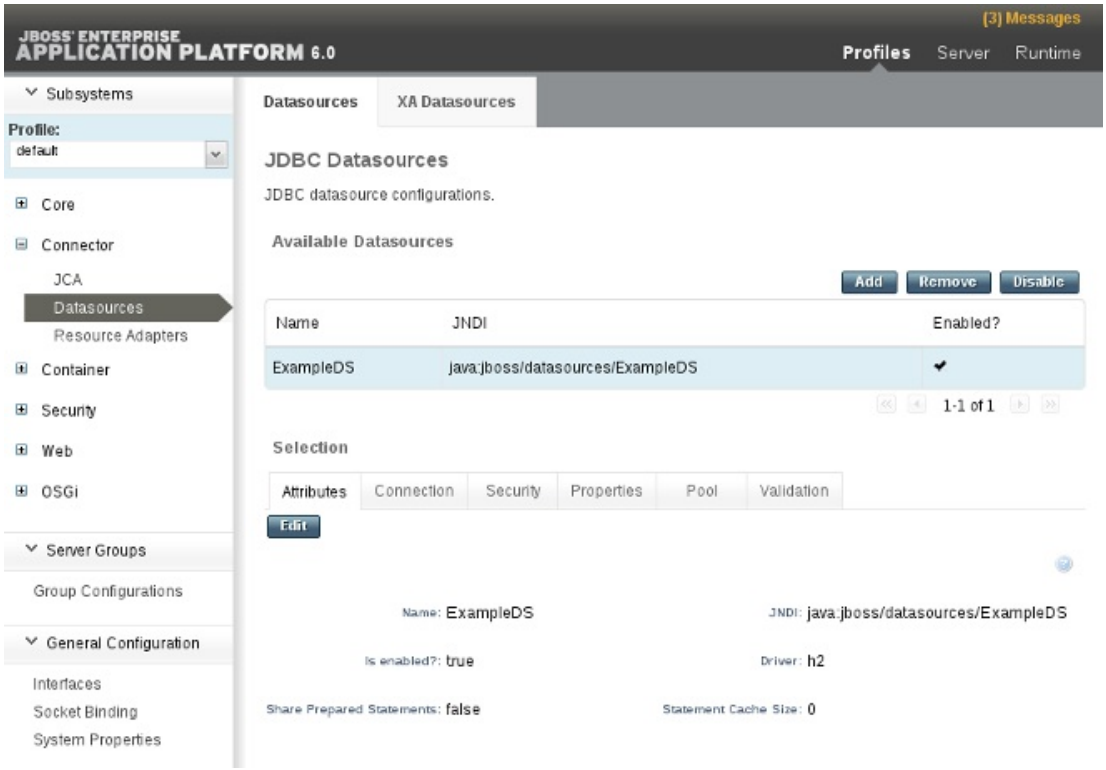


図10.2 データソースパネル

c. **新しいデータソースの作成**

i. **[Datasources]** パネル上部にある **[Add]** ボタンを選択します。

ii. **Create Datasource** ウィザードで新しいデータソースの属性を入力し、**[Next]** ボタンを押します。

iii. **Create Datasource** ウィザードで JDBC ドライバーの詳細を入力し、**[Next]** ボタンを押します。

iv. **Create Datasource** ウィザードで接続設定を入力し、**[Done]** ボタンを押します。

結果

非 XA データソースがサーバーに追加されます。**standalone.xml** または **domain.xml** ファイル、および管理インターフェースで追加を確認することができます。

バグを報告する

10.6.2.5. データソースのパラメーター

表10.1 非 XA および XA データソースに共通のデータソースパラメーター

パラメーター	説明
jndi-name	データソースの一意の JNDI 名。
pool-name	データソースの管理プール名。

パラメーター	説明
enabled	データソースが有効かどうかを指定します。
use-java-context	データソースをグローバルの JNDI にバインドするかどうかを指定します。
spy	JDBC レイヤーで **spy** 機能を有効にします。この機能は、データソースへの JDBC トラフィックをすべてログに記録します。また、**logging-category** パラメーターを **org.jboss.jdbc** に設定する必要があります。
use-ccm	キャッシュ接続マネージャーを有効にします。
new-connection-sql	接続プールに接続が追加された時に実行する SQL ステートメント。
transaction-isolation	次のいずれかになります。 • TRANSACTION_READ_UNCOMMITTED • TRANSACTION_READ_COMMITTED • TRANSACTION_REPEATABLE_READ • TRANSACTION_SERIALIZABLE • TRANSACTION_NONE
url-delimiter	高可用性 (HA) クラスターデータベースの connection-url にある URL の区切り文字。
url-selector-strategy-class-name	インターフェース **org.jboss.jca.adapters.jdbc.URLSelectorStrategy** を実装するクラス。
security	セキュリティー設定である子要素が含まれます。表 10.6「セキュリティーパラメーター」 を参照してください。
validation	検証設定である子要素が含まれます。表10.7「検証パラメーター」 を参照してください。
timeout	タイムアウト設定である子要素が含まれます。表 10.8「タイムアウトパラメーター」 を参照してください。
statement	ステートメント設定である子要素が含まれます。表 10.9「ステートメントのパラメーター」 を参照してください。

表10.2 非 XA データソースのパラメーター

パラメーター	説明
jta	非 XA データソースの JTA 統合を有効にします。XA データソースには適用されません。
connection-url	JDBC ドライバーの接続 URL。
driver-class	JDBC ドライバークラスの完全修飾名。
connection-property	**`Driver.connect(url,props)`** メソッドに渡される任意の接続プロパティー。各 connection-property は、文字列名と値のペアを指定します。プロパティー名は名前、値は要素の内容に基づいています。
pool	プーリング設定である子要素が含まれます。表 10.4「非 XA および XA データソースに共通のプールパラメーター」 を参照してください。

表10.3 XA データソースのパラメーター

パラメーター	説明
xa-datasource-property	実装クラス **`XADataSource`** に割り当てるプロパティー。*name=value* で指定。 **`setName`** という形式で setter メソッドが存在する場合、プロパティーは **`setName(value)`** という形式の setter メソッドを呼び出すことで設定されます。
xa-datasource-class	実装クラス **`javax.sql.XADataSource`** の完全修飾名。
driver	JDBC ドライバーが含まれるクラスローダーモジュールへの一意参照。*driverName#majorVersion.minorVersion* の形式にのみ対応しています。
xa-pool	プーリング設定である子要素が含まれます。表 10.4「非 XA および XA データソースに共通のプールパラメーター」 と 表10.5「XA プールパラメーター」 を参照してください。
recovery	リカバリ設定である子要素が含まれます。表 10.10「リカバリーパラメーター」 を参照してください。

表10.4 非 XA および XA データソースに共通のプールパラメーター

パラメーター	説明
min-pool-size	プールが保持する最小接続数

パラメーター	説明
max-pool-size	プールが保持可能な最大接続数
prefill	接続プールのプレフィルを試行するかどうかを指定します。要素が空の場合は **true** を示します。デフォルトは、**false** です。
use-strict-min	pool-size が厳密かどうかを指定します。デフォルトは **false** に設定されています。
flush-strategy	エラーの場合にプールをフラッシュするかどうかを指定します。有効な値は次のとおりです。 • FailingConnectionOnly • IdleConnections • EntirePool デフォルトは **FailingConnectionOnly** です。
allow-multiple-users	複数のユーザーが getConnection (user, password) メソッドを使いデータソースへアクセスするか、また内部プールタイプがこの動作に対応するかを指定します。

表10.5 XA プールパラメーター

パラメーター	説明
is-same-rm-override	**javax.transaction.xa.XAResource.isSameRM(XAResource)** クラスが**true** あるいは **false** のどちらを返すかを指定します。
interleaving	XA 接続ファクトリのインターリービングを有効にするかどうかを指定します。
no-tx-separate-pools	コンテキスト毎に sub-pool を作成するかどうかを指定します。これには Oracle のデータソースが必要ですが、このデータソースは JTA トランザクションの内部、外部に関わらず、XA 接続の利用ができなくなります。
pad-xid	Xid のパディングを行うかどうかを指定します。
wrap-xa-resource	XAResource を **org.jboss.tm.XAResourceWrapper** インスタンスでラップするかどうかを指定します。

表10.6 セキュリティーパラメーター

パラメーター	説明
user-name	新規接続の作成に使うユーザー名
password	新規接続の作成に使うパスワード
security-domain	認証処理を行う JAAS security-manager 名が含まれます。この名前は、JAAS ログイン設定の application-policy/name 属性に相関します。
reauth-plugin	物理接続の再認証に使う再認証プラグインを定義します。

表10.7 検証パラメーター

パラメーター	説明
valid-connection-checker	**SQLException.isValidConnection(Connection e)** メソッドを提供し接続を検証するインターフェース **org.jboss.jca.adaptors.jdbc.ValidConnectionChecker** の実装。例外が発生すると接続が破棄されます。存在する場合、**check-valid-connection-sql** パラメーターが上書きされます。
check-valid-connection-sql	プール接続の妥当性を確認する SQL ステートメント。これは、管理接続をプールから取得し利用する場合に呼び出される場合があります。
validate-on-match	接続ファクトリが指定のセットに対して管理対象接続をマッチしようとした時に接続レベルの検証を実行するかどうかを示します。 通常、**validate-on-match** に true を指定した時に **background-validation** を true に指定することはありません。クライアントが使用する前に接続を検証する必要がある場合に **Validate-on-match** が必要になります。このパラメーターはデフォルトでは true になっています。
background-validation	接続がバックグラウンドスレッドで検証されることを指定します。**validate-on-match** を使用しない場合、バックグラウンドの検証はパフォーマンスを最適化します。**validate-on-match** が true の時に **background-validation** を使用すると、チェックが冗長になることがあります。バックグラウンド検証では、不良の接続がクライアントに提供される可能性があります (検証スキャンと接続がクライアントに提供されるまでの間に接続が悪くなります)。そのため、クライアントアプリケーションはこの接続不良の可能性に対応する必要があります。

パラメーター	説明
background-validation-millis	バックグラウンド検証を実行する期間 (ミリ秒単位)。
use-fast-fail	true の場合、接続が無効であれば最初に接続を割り当てしようとした時点で失敗します。デフォルトは **false** です。
stale-connection-checker	ブール値の **isStaleConnection(SQLException e)** メソッドを提供する **org.jboss.jca.adapters.jdbc.StaleConnectionChecker** のインスタンス。このメソッドが **true** を返すと、**SQLException** のサブクラスである **org.jboss.jca.adapters.jdbc.StaleConnectionException** に例外がラップされます。
exception-sorter	ブール値である **isExceptionFatal(SQLException e)** メソッドを提供する **org.jboss.jca.adapters.jdbc.ExceptionSorter** のインスタンス。このメソッドは、例外が **connectionErrorOccurred** メッセージとして **javax.resource.spi.ConnectionEventListener** のすべてのインスタンスへブロードキャストされるべきであるかどうかを検証します。

表10.8 タイムアウトパラメーター

パラメーター	説明
use-try-lock	**lock()** の代わりに **tryLock()** を使用します。これは、ロックが使用できない場合に即座に失敗するのではなく、設定された秒数間ロックの取得を試みます。デフォルトは **60** 秒です。たとえば、タイムアウトを 5 分に設定するには、**<use-try-lock>**300**</use-try-lock>** を設定します。
blocking-timeout-millis	接続待機中にブロックする最大時間 (ミリ秒)。この時間を超過すると、例外がスローされます。これは、接続許可の待機中のみブロックし、新規接続の作成に長時間要している場合は例外をスローしません。デフォルトは 30000 (30 秒) です。
idle-timeout-minutes	アイドル接続が切断されるまでの最大時間 (分単位)。実際の最大時間は idleRemover のスキャン時間によって異なります。idleRemover のスキャン時間はプールの最小 **idle-timeout-minutes** の半分になります。

パラメーター	説明
set-tx-query-timeout	トランザクションがタイムアウトするまでの残り時間を基にクエリのタイムアウトを設定するかどうかを指定します。トランザクションが存在しない場合は設定済みのクエリのタイムアウトが使用されます。デフォルトは **false** です。
query-timeout	クエリのタイムアウト (秒)。デフォルトはタイムアウトなしです。
allocation-retry	例外をスローする前に接続の割り当てを再試行する回数。デフォルトは **0** で、初回の割り当て失敗で例外がスローされます。
allocation-retry-wait-millis	接続の割り当てを再試行するまで待機する期間 (ミリ秒単位)。デフォルトは 5000 (5 秒) です。
xa-resource-timeout	ゼロでない場合、この値は **XAResource.setTransactionTimeout** メソッドへ渡されます。

表10.9 ステートメントのパラメーター

パラメーター	説明
track-statements	接続がプールへ返され、ステートメントが準備済みステートメントキャッシュへ返された時に、閉じられていないステートメントをチェックするかどうかを指定します。false の場合、ステートメントは追跡されません。 **有効な値** • **true**: ステートメントと結果セットが追跡され、ステートメントが閉じられていない場合は警告が出力されます。 • **false**: ステートメントと結果セットのいずれも追跡されません。 • **nowarn**: ステートメントは追跡されますが、警告は出力されません。これがデフォルト設定となっています。
prepared-statement-cache-size	LRU (Least Recently Used) キャッシュにある接続毎の準備済みステートメントの数。
share-prepared-statements	閉じずに同じステートメントを 2 回要求した場合に、同じ基盤の準備済みステートメントを使用するかどうかを指定します。デフォルトは **false** です。

表10.10 リカバリーパラメーター

パラメーター	説明
recover-credential	リカバリーに使用するユーザー名とパスワードのペア、あるいはセキュリティドメイン。
recover-plugin	リカバリーに使用される **`org.jboss.jca.core.spi.recovery.RecoveryPlugin`** クラスの実装。

バグを報告する

10.6.3. トランザクションロギング

10.6.3.1. トランザクションログメッセージについて

ログファイルが読み取り可能な状態でトランザクションの状態を追跡するには、トランザクションロガーに **DEBUG** ログレベルを使用します。詳細なデバッグでは **TRACE** ログレベルを使用します。トランザクションロガーの設定に関する詳細は 「トランザクションサブシステムのログ設定」を参照してください。

TRACE ログレベルに設定すると、トランザクションマネージャーは多くのロギング情報を生成できます。一般的に表示されるメッセージの一部は次のとおりです。他のメッセージが表示されることもあります。

表10.11 トランザクションステートの変更

トランザクションの開始	トランザクションが開始されると、次のコードが実行されます。 `com.arjuna.ats.arjuna.coordinator.BasicAction::Begin:1342` `tsLogger.logger.trace("BasicAction::Begin() for action-id "+ get_uid());`
トランザクションのコミット	トランザクションがコミットすると、次のコードが実行されます。 `com.arjuna.ats.arjuna.coordinator.BasicAction::End:1342` `tsLogger.logger.trace("BasicAction::End() for action-id "+ get_uid());`

トランザクションのロールバック	トランザクションがロールバックすると、次のコードが実行されます。 ``` com.arjuna.ats.arjuna.coordinator .BasicAction::Abort:1575 ``` ``` tsLogger.logger.trace("BasicActio n::Abort() for action-id "+ get_uid()); ```
トランザクションのタイムアウト	トランザクションがタイムアウトすると、次のコードが実行されます。 ``` com.arjuna.ats.arjuna.coordinator .TransactionReaper::doCancellatio ns:349 ``` ``` tsLogger.logger.trace("Reaper Worker " + Thread.currentThread() + " attempting to cancel " + e._control.get_uid()); ``` その後、上記のように同じスレッドがトランザクションをロールバックすることが確認できます。

バグを報告する

10.6.3.2. トランザクションサブシステムのログ設定

概要

JBoss Enterprise Application Platform の他のログ設定に依存せずにトランザクションログの情報量を制御する手順を説明します。主に Web ベースの管理コンソールを用いた手順を説明し、管理 CLI のコマンドはその説明の後で取り上げます。

手順10.4 管理コンソールを使用したトランザクションロガーの設定

1. **ログ設定エリアへの移動**
   管理コンソールにて画面の左上にある **[Profiles]** タブをクリックします。管理対象ドメインを使用する場合は、右上の **[Profile]** 選択ボックスから設定したいサーバープロファイルを選択します。

   [Core] メニューを展開して、**[Logging]** ラベルをクリックします。

2. **`com.arjuna` 属性を編集します。**
   ページの下の方にある **[Details]** セクションの **[Edit]** ボタンをクリックします。ここにクラス固有のログ情報を追加できます。**`com.arjuna`** クラスはすでに存在しています。ログレベルと親ハンドラーを使用するかどうか変更できます。

   ログレベル

デフォルトのログレベルは **WARN** です。トランザクションはログを大量に出力できるため、標準的なログレベルの意味は、トランザクションロガーでは若干異なります。通常、選択したレベルより重要度が低いレベルでタグ付けされたメッセージは破棄されます。

トランザクションログのレベル (詳細度が最高レベルから最低レベルまで)

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- FAILURE

親ハンドラーの使用

ロガーがログ出力を親ロガーに送信するかどうかを指定します。デフォルトの動作は **true** です。

3. 変更は直ちに反映されます。

バグを報告する

10.6.3.3. トランザクションの参照と管理

コマンドラインベースの管理 CLI では、トランザクションレコードを参照および操作する機能がサポートされます。この機能は、トランザクションマネージャーと JBoss Enterprise Application Platform 6 の管理 API との対話によって提供されます。

トランザクションマネージャーは、待機中の各トランザクションとトランザクションに関連する参加者に関する情報を、*オブジェクトストア*と呼ばれる永続ストレージに格納します。管理 API は、オブジェクトストアを **log-store** と呼ばれるリソースとして公開します。**probe** と呼ばれる API 操作はトランザクションログを読み取り、各ログに対してノードを作成します。**probe** コマンドは、**log-store** を更新する必要があるときに、いつでも手動で呼び出すことができます。トランザクションログが現れて、すぐに消失されるのは通常のことです。

例10.1 ログストアの更新

このコマンドは、管理対象ドメインでプロファイル **default** を使用するサーバーグループに対してログストアを更新します。スタンドアローンサーバーの場合は、コマンドから **profile=default** を削除します。

```
/profile=default/subsystem=transactions/log-store=log-store/:probe
```

例10.2 準備されたすべてのトランザクションの表示

準備されたすべてのトランザクションを表示するには、最初にログストアを更新し (例10.1「ログストアの更新」 を参照)、ファイルシステムの **ls** コマンドに類似した機能を持つ次のコマンドを実行します。

```
ls /profile=default/subsystem=transactions/log-store=log-
store/transactions
```

各トランザクションが一意の ID とともに表示されます。個々の操作は、各トランザクションに対して実行できます (トランザクションの管理 を参照)。

トランザクションの管理

トランザクションの属性を表示します。

JNDI 名、EIS 製品名およびバージョン、ステータスなどのトランザクションに関する情報を表示するには、**:read-resource** CLIコマンドを使用します。

```
/profile=default/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9:read-resource
```

トランザクションの参加者を表示します。

各トランザクションログには、**participants** (参加者) と呼ばれる子要素が含まれます。トランザクションの参加者を確認するには、この要素に対して **read-resource** CLI コマンドを使用します。参加者は、JNDI 名によって識別されます。

```
/profile=default/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-
b66efc2\:4f9e6f8f\:9/participants=java\:\/JmsXA:read-resource
```

結果は以下のようになります。

```
{
   "outcome" => "success",
   "result" => {
       "eis-product-name" => "HornetQ",
       "eis-product-version" => "2.0",
       "jndi-name" => "java:/JmsXA",
       "status" => "HEURISTIC",
       "type" => "/StateManager/AbstractRecord/XAResourceRecord"
   }
}
```

ここで示された結果ステータスは **HEURISTIC** であり、リカバリーが可能です。詳細については、「トランザクションをリカバリーします。」を参照してください。

トランザクションを削除します。

各トランザクションログは、トランザクションを表すトランザクションログを削除するために、**:delete** 操作をサポートします。

```
/profile=default/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9:delete
```

トランザクションをリカバリーします。

各トランザクションログは、**:recover** CLI コマンドを使用したリカバリーをサポートします。

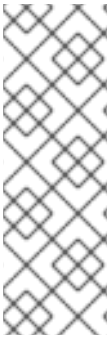
ヒューリスティックなトランザクションと参加者のリカバリー

- トランザクションのステータスが **HEURISTIC** である場合は、リカバリー操作によって、ステータスが **PREPARE** に変わり、リカバリーがトリガーされます。
- トランザクションの参加者の 1 つがヒューリスティックな場合、リカバリー操作により、**commit** 操作の応答が試行されます。成功した場合、トランザクションログから参加者が削除されます。これを確認するには、**log-store** 上で **:probe** 操作を再実行し、参加者がリストされていないことを確認します。これが最後の参加者の場合は、トランザクションも削除されます。

リカバリーが必要なトランザクションのステータスを更新します。

トランザクションをリカバリーする必要がある場合は、リカバリーを試行する前に **:refresh** CLI コマンドを使用して、トランザクションのリカバリーが必要であるかを確認できます。

```
/profile=default/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9:refresh
```



注記

JTS トランザクションで、参加者がリモートサーバー上にある場合、トランザクションマネージャーは制限された量の情報のみ使用できることがあります。この場合は、HornetQ ストレージモードではなくファイルベースのオブジェクトストアを使用することが推奨されます。HornetQ ストレージモードを使用するには、トランザクションマネージャーの **use-hornetq-store** オプションの値を **true** に設定します。トランザクションマネージャーの設定については、「トランザクションマネージャーの設定」を参照してください。

トランザクション統計情報の表示

トランザクションマネージャー (TM) の統計が有効になっていると、トランザクションマネージャーおよびトランザクションサブシステムに関する統計を表示できます。TM の統計を有効にする方法は 「トランザクションマネージャーの設定」 を参照してください。

統計は、Web ベースの管理コンソールまたはコマンドラインの管理 CLI より表示できます。Web ベースの管理コンソールでは、トランザクション統計情報は **[Runtime]** → **[Subsystem Metrics]** → **[Transactions]** を選択して取得できます。トランザクション統計情報は、管理対象ドメインの各サーバーでも利用できます。左上にある **[Server]** 選択ボックスで、サーバーを指定できます。

以下の表は、利用可能な各統計情報、その説明、および統計情報を表示する CLI コマンドを示しています。

表10.12 トランザクションサブシステム統計情報

統計	説明	CLI コマンド

統計	説明	CLI コマンド
Total (合計)	このサーバー上でトランザクションマネージャーにより処理されるトランザクションの合計数。	`/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-transactions,include-defaults=true)`
Committed (コミット済み)	このサーバー上でトランザクションマネージャーにより処理されるコミット済みトランザクションの数。	`/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-committed-transactions,include-defaults=true)`
Aborted (異常終了)	このサーバー上でトランザクションマネージャーにより処理される異常終了したトランザクションの数。	`/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-aborted-transactions,include-defaults=true)`
Timed Out (タイムアウト)	このサーバー上でトランザクションマネージャーにより処理されるタイムアウトのトランザクションの数。	`/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-timed-out-transactions,include-defaults=true)`

統計	説明	CLI コマンド
Heuristics （ヒューリスティック）	管理コンソールで利用不可です。ヒューリスティック状態のトランザクションの数。	`/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-heuristics,include-defaults=true)`
In-Flight Transactions (フライト状態のトランザクション)	管理コンソールでは使用できません。開始した未終了のトランザクションの数。	`/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-inflight-transactions,include-defaults=true)`
Failure Origin - Applications (障害の原因 - アプリケーション)	障害の原因がアプリケーションであった失敗トランザクションの数。	`/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-application-rollbacks,include-defaults=true)`
Failure Origin - Resources (障害の原因 - リソース)	障害の原因がリソースであった失敗トランザクションの数。	`/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-resource-rollbacks,include-defaults=true)`

バグを報告する

10.7. JTA トランザクションの使用

10.7.1. トランザクション JTA タスクの概要

はじめに

次の手順は、アプリケーションでトランザクションを使用する必要がある場合に役に立ちます。

- 「トランザクションの制御」
- 「トランザクションの開始」
- 「トランザクションのコミット」
- 「トランザクションのロールバック」
- 「トランザクションにおけるヒューリスティックな結果の処理方法」
- 「トランザクションマネージャーの設定」
- 「トランザクションエラーの処理」

バグを報告する

10.7.2. トランザクションの制御

はじめに

この手順のリストでは、JTA または JTS API を使用するアプリケーションでトランザクションを制御するさまざまな方法を概説します。

- 「トランザクションの開始」
- 「トランザクションのコミット」
- 「トランザクションのロールバック」
- 「トランザクションにおけるヒューリスティックな結果の処理方法」

バグを報告する

10.7.3. トランザクションの開始

この手順では、Java Transaction Service (JTS) プロトコルを使用して、新しい JTA トランザクションを開始する方法、または分散トランザクションに散開する方法を示します。

分散トランザクション

分散トランザクションでは、トランザクション参加者が複数のサーバー上の個別アプリケーションに存在します。参加者が新しいトランザクションコンテキストを作成する代わりに、すでに存在するトランザクションに参加する場合は、コンテキストを共有する 2 人以上の参加者が分散トランザクションに参加します。分散トランザクションを使用するには、ORB　を設定する必要があります。ORB 設定の詳細については、『管理および設定ガイド』の項 『ORB 設定』を参照してください。

1. **`UserTransaction` のインスタンスを取得します。**
   `@TransactionManagement(TransactionManagementType.BEAN)` アノテーションを用いると、 JNDI やインジェクション (EJB が Bean 管理のトランザクションを使用する場合は EJB の EjbContext) を使用してインスタンスを取得できます。

   - **JNDI**

     ```
     new InitialContext().lookup("java:comp/UserTransaction")
     ```

- インジェクション

```
@Resource UserTransaction userTransaction;
```

- **EjbContext**

```
EjbContext.getUserTransaction()
```

2. **データソースに接続後、`UserTransaction.begin()` を呼び出します。**

```
...
try {
    System.out.println("\nCreating connection to database: "+url);
    stmt = conn.createStatement();  // non-tx statement
    try {
        System.out.println("Starting top-level transaction.");
        userTransaction.begin();
        stmtx = conn.createStatement(); // will be a tx-statement
        ...
    }
}
```

JTS API を使用して既存のトランザクションに参加します。

EJB の利点の 1 つは、コンテナがすべてのトランザクションを管理することです。ORB をセットアップした場合は、コンテナによって分散トランザクションが管理されます。

結果

トランザクションが開始します。トランザクションをコミットまたはロールバックするまで、データソースのすべての使用でトランザクションに対応します。

注記

全体の例は 「JTA トランザクションの例」を参照してください。

バグを報告する

10.7.4. トランザクションのネスト

ネストトランザクションは、JTS API による分散トランザクションを使用する場合のみサポートされます。また、多くのデータベースベンダーはネストトランザクションをサポートしないため、ネストトランザクションをアプリケーションに追加する前に、データベースベンダーにお問い合わせください。

OTS 仕様では、ネストトランザクションのタイプが制限されます。サブトランザクションコミットプロトコルは最上位トランザクションと同じです。**`prepare`** フェーズと **`commit`** フェーズまたは **`abort`** フェーズの 2 つのフェーズがあります。このようなネストトランザクションを行うと、サブトランザクションコーディネーターがコミット中にリソースがコミットできないことを検出するなどの、不整合な結果が生じることがあります。コーディネーターはコミットされたリソースを中止するよう指示できず、ヒューリスティックな結果が生じます。この厳密な OTS のネストトランザクションは、**`CosTransactions::SubtransactionAwareResource`** インターフェースを介して利用できます。

JTS の JBoss Enterprise Application Platform の実装はこのタイプのネストトランザクションをサポートします。また、厳密な OTS モデルで発生する可能性がある問題を回避するマルチフェーズコ

ミットプロトコルを使用するネストトランザクションのタイプもサポートします。このタイプのネストトランザクションは **ArjunaOTS::ArjunaSubtranAwareResource** を介して利用でき、ネストトランザクションをコミットするたびに 2 相コミットプロトコルにより駆動されます。

ネストトランザクションを作成するには、親トランザクション内で新しいトランザクションを作成します。トランザクションの作成については、「トランザクションの開始」を参照してください。

ネストトランザクションの効果は、エンクローズトランザクションのコミットまたはロールバックに依存します。エンクローズトランザクションが中止された場合は、ネストトランザクションがコミットされた場合であっても効果はリカバリされます。

バグを報告する

10.7.5. トランザクションのコミット

この手順では、Java Transaction API (JTA) を使用してトランザクションをコミットする方法を示します。この API は、ローカルトランザクションと分散トランザクションの両方に使用されます。分散トランザクションは、Java Transaction Server (JTS) により管理され、Object Request Broker (ORB) の設定を必要とします。ORB の設定の詳細については、『管理および設定ガイド』の項 『ORB 設定』を参照してください。

要件

トランザクションは、コミットする前に開始する必要があります。トランザクションの開始方法については、「トランザクションの開始」を参照してください。

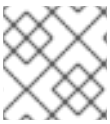
1. **UserTransaction の commit() メソッドを呼び出します。**
   UserTransaction の **commit()** メソッドを呼び出すと、トランザクションマネージャーがトランザクションをコミットしようとします。

```
@Inject
private UserTransaction userTransaction;

public void updateTable(String key, String value)
    EntityManager entityManager =
entityManagerFactory.createEntityManager();
    try {
        userTransaction.begin():
        <!-- Perform some data manipulation using entityManager -->
     ...
     // Commit the transaction
        userTransaction.commit();
    } catch (Exception ex) {
        <!-- Log message or notify Web page -->
        ...
        try {
            userTransaction.rollback();
        } catch (SystemException se) {
            throw new RuntimeException(se);
        }
        throw new RuntimeException(e);
    } finally {
        entityManager.close();
    }
}
```

2. **Container Managed Transactions (CMT) を使用する場合は、手動でコミットする必要がありません。**
   Bean が Container Managed Transactions を使用するよう設定すると、コンテナはコードで設定したアノテーションに基づいてトランザクションライフサイクルを管理します。

結果

データソースがコミットし、トランザクションが終了します。そうでない場合は、例外がスローされます。



注記

全体の例は 「JTA トランザクションの例」 を参照してください。

バグを報告する

10.7.6. トランザクションのロールバック

この手順では、Java Transaction API (JTA) を使用してトランザクションをロールバックする方法を示します。この API は、ローカルトランザクションと分散トランザクションの両方に使用されます。分散トランザクションは、Java Transaction Server (JTS) により管理され、Object Request Broker (ORB) の設定を必要とします。ORB の設定の詳細については、『管理および設定ガイド』の項 『ORB 設定』 を参照してください。

要件

トランザクションは、ロールバックする前に開始する必要があります。トランザクションの開始方法については、「トランザクションの開始」を参照してください。

1. **`UserTransaction` の `rollback()` メソッドを呼び出します。**
   `UserTransaction` の **`rollback()`** メソッドを呼び出すと、トランザクションマネージャーがトランザクションをロールバックし、データを前の状態に戻そうとします。

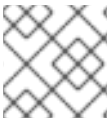
```
@Inject
private UserTransaction userTransaction;

public void updateTable(String key, String value)
    EntityManager entityManager =
entityManagerFactory.createEntityManager();
    try {
        userTransaction.begin():
        <!-- Perform some data manipulation using entityManager -->
     ...
     // Commit the transaction
        userTransaction.commit();
    } catch (Exception ex) {
        <!-- Log message or notify Web page -->
        ...
        try {
            userTransaction.rollback();
        } catch (SystemException se) {
            throw new RuntimeException(se);
        }
        throw new RuntimeException(e);
    } finally {
```

```
            entityManager.close();
        }
    }
```

2. **Container Managed Transactions (CMT) を使用する場合は、手動でトランザクションをロールバックする必要がありません。**
   Bean が Container Managed Transactions を使用するよう設定すると、コンテナはコードで設定したアノテーションに基づいてトランザクションライフサイクルを管理します。

結果

トランザクションマネージャーにより、トランザクションがロールバックされます。



注記

全体の例は「JTA トランザクションの例」を参照してください。

バグを報告する

10.7.7. トランザクションにおけるヒューリスティックな結果の処理方法

この手順では、Java Transaction Service (JTS) を使用して JTA トランザクション (ローカルまたは分散) でヒューリスティックな結果を処理する方法を示します。分散トランザクションを使用する場合は、ORB を設定する必要があります。ORB 設定の詳細については、『管理および設定ガイド』の項『ORB 設定』を参照してください。

ヒューリスティックな結果はよく発生するものではなく、通常は例外的な原因があります。*ヒューリスティック*という言葉は「手動」を意味し、こうした結果が通常処理される方法です。トランザクションのヒューリスティックな結果については、「ヒューリスティックな結果について」を参照してください。

手順10.5 トランザクションでのヒューリスティックな結果の処理方法

1. **原因を突き止める**
   トランザクションのヒューリスティックな結果の全体的な原因は、リソースマネージャーがコミットまたはロールバックの実行を約束したにも関わらず、失敗したことにあります。原因としては、サードパーティーコンポーネント、サードパーティーコンポーネントと JBoss Enterprise Application Platform 間の統合レイヤー、JBoss Enterprise Application Platform 自体に問題がある可能性があります。

   ヒューリスティックなエラーの最も一般的な原因として圧倒的に多いのが、環境の一時的な障害とリソースマネージャーを扱うコードのコーディングエラーの 2 つです。

2. **環境内の一時的な障害を修正する**
   通常、環境内で一時的な障害が発生した場合は、ヒューリスティックなエラーを発見する前に気づきます。原因としては、ネットワークの停止、ハードウェア障害、データベース障害、電源異常、その他多くの可能性があります。

   ストレステストの実施中にテスト環境でヒューリスティックな結果が発生した場合は、使用している環境の脆弱性に関する情報が提供されます。

警告

JBoss Enterprise Application Platform は、障害発生時に非ヒューリスティックな状態にあるトランザクションの自動回復を行いますが、ヒューリスティックなトランザクションの回復は試行しません。

3. **リソースマネージャーのベンダーに連絡する**
   明らかに使用している環境に障害がない場合や、ヒューリスティックな結果が容易に再現可能な場合は、コーディングエラーである可能性があります。サードパーティーのベンダーに連絡して、解決方法の有無を確認してください。JBoss Enterprise Application Platform のトランザクションマネージャー自体に問題があると思われる場合は、Red Hat グローバルサポートサービスにご連絡ください。

4. **テスト環境の場合は、ログを削除して JBoss Enterprise Application Platform を再起動する**
   テスト環境である場合や、データの整合性を気にしない場合は、トランザクションログを削除して JBoss Enterprise Application Platform を再起動すると、ヒューリスティックな結果はなくなります。デフォルトのトランザクションログの場所はスタンドアロンサーバーでは ***EAP_HOME*/standalone/data/tx-object-store/**、管理ドメインでは ***EAP_HOME*/domain/servers/*SERVER_NAME*/data/tx-object-store** になります。管理ドメインの *SERVER_NAME* は、サーバーグループに参加している個々のサーバー名になります。

5. **手作業で結果を解決する**
   トランザクションの結果を手作業で解決するプロセスは、障害の厳密な状況によって大きく左右されます。通常は、以下の手順に従い、それぞれの状況に適用してください。

   a. 関連するリソースマネージャーを特定する。

   b. トランザクションマネージャーの状態とリソースマネージャーを調べる。

   c. 関与する 1 つ以上のコンポーネント内でログのクリーンアップとデータ調整を手動で強制する。

   これらの手順を実行する方法の詳細は、本書の範囲外となります。

バグを報告する

10.7.8. トランザクションのタイムアウト

10.7.8.1. トランザクションタイムアウトについて

原子性を確保し、トランザクションを ACID 標準に準拠させるため、トランザクションの一部が長期間実行される場合があります。トランザクションの参加者は、コミット時にデータソースの一部をロックする必要があります。また、トランザクションマネージャーは各トランザクション参加者からの応答を待ってからすべての参加者にコミットあるいはロールバックの指示を出す必要があります。ハードウェアあるいはネットワークの障害のため、リソースが永久にロックされることがあります。

トランザクションのタイムアウトをトランザクションと関連付け、ライフサイクルを制御することができます。タイムアウトのしきい値がトランザクションのコミットあるいはロールバック前に渡された場合、タイムアウトにより、自動的にトランザクションがロールバックされます。

トランザクションサブシステム全体に対しデフォルトのタイムアウト値を設定できます。または、デフォルトのタイムアウト値を無効にし、トランザクションごとにタイムアウトを指定できます。

バグを報告する

10.7.8.2. トランザクションマネージャーの設定

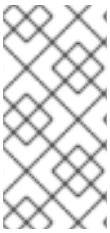
トランザクションマネージャー (TM) は、Web ベースの管理コンソールかコマンドラインの管理 CLI を使用して設定できます。各コマンドやオプションでは、 JBoss Enterprise Application Platform を管理対象ドメインとして実行していると仮定します。スタンドアロンサーバーを使用する場合や **default** 以外のプロファイルを修正したい場合は、以下の方法で手順とコマンドを修正する必要があることがあります。

例のコマンドに関する注意点

- 管理コンソールの場合、**default** プロファイルは最初のコンソールログイン時に選択されるものです。異なるプロファイルでトランザクションマネージャーの設定を修正する必要がある場合は、**default** の代わりに使用しているプロファイルを選択してください。

  同様に、例の CLI コマンドの **default** プロファイルを使用しているプロファイルに置き換えてください。

- スタンドアロンサーバーを使用する場合、存在するプロファイルは 1 つのみです。特定のプロファイルを選択する手順は無視してください。CLI コマンドでは、例のコマンドの **/profile=default** 部分を削除してください。



注記

TM オプションが管理コンソールまたは管理 CLI で表示されるようにするには、**transactions** サブシステムが有効でなくてはなりません。これは、デフォルトで有効になっており、他の多くのサブシステムが適切に機能するために必要なため、無効にする可能性は大変低くなります。

管理コンソールを使用した TM の設定

Web ベースの管理コンソールを使用して TM を設定するには、管理コンソール画面の左上にある一覧から **[Runtime]** タブを選択します。管理対象ドメインを使用する場合、選択できるプロファイルがいくつかあります。プロファイル画面の右上にある **[Profile]** 選択ボックスから適切なプロファイルを選択してください。**[Container]** メニューを展開して、**[Transactions]** を選択します。

トランザクションマネージャーの設定ページには、さらなるオプションが表示されています。**[Recovery]** オプションはデフォルトでは非表示です。展開するには、**[Recovery]** ヘッダーをクリックします。オプションを編集するには、**[Edit]** ボタンをクリックします。変更は直ちに反映されます。

インラインヘルプを表示するには、**[Need Help?]** ラベルをクリックします。

管理 CLI を使用した TM の設定

管理 CLI では、一連のコマンドを使用して TM を設定できます。プロファイル **default** の管理対象ドメインの場合、コマンドはすべて **/profile=default/subsystem=transactions/** で始まり、スタンドアロンサーバーの場合は **/subsystem=transactions** で始まります。

表10.13 TM 設定オプション

オプション	説明	CLI コマンド
統計の有効化 (Enable Statistics)	トランザクションの統計を有効にするかどうか指定します。統計は **Runtime** タブの **Subsystem Metrics** セクションにある管理コンソールで閲覧できます。	**/profile=default/subsystem=transactions/:write-attribute(name=enable-statistics,value=true)**
TSM ステータスの有効化 (Enable TSM Status)	トランザクションステータスマネージャー (TSM) のサービスを有効にするかどうか指定します。これは、アウトオブプロセスのリカバリに使用されます。	**/profile=default/subsystem=transactions/:write-attribute(name=enable-tsm-status,value=false)**
デフォルトのタイムアウト (Default Timeout)	デフォルトのトランザクションタイムアウトです。デフォルトでは **300** 秒に設定されています。トランザクションごとにプログラムで上書きできます。	**/profile=default/subsystem=transactions/:write-attribute(name=default-timeout,value=300)**
パス (Path)	トランザクションマネージャーコアがデータを格納するファイルシステムの相対または絶対パスです。デフォルトの値は **relative-to** 属性の値と相対的なパスです。	**/profile=default/subsystem=transactions/:write-attribute(name=path,value=var)**
相対的 (Relative To)	ドメインモデルのグローバルなパス設定を参照します。デフォルト値は、JBoss Enterprise Application Platform 6 のデータディレクトリで、**jboss.server.data.dir** プロパティーの値です。デフォルトは、管理対象ドメインの場合は ***EAP_HOME*/domain/data/**、スタンドアロンサーバーインスタンスの場合は ***EAP_HOME*/standalone/data/** です。**path** TM 属性の値は、このパスに相対的です。空の文字列を使用して、デフォルト動作を無効にし、**path** 属性の値が絶対パスとして強制的に扱われるようにします。	**/profile=default/subsystem=transactions/:write-attribute(name=relative-to,value=jboss.server.data.dir)**
オブジェクトストアパス (Object Store Path)	TM オブジェクトストアがデータを格納するファイルシステムの相対または絶対パスです。デフォルトでは、**object-store-relative-to** パラメーターの値に相対的です。	**/profile=default/subsystem=transactions/:write-attribute(name=object-store-path,value=tx-object-store)**

オプション	説明	CLI コマンド
オブジェクトストアパスに相対的 (Object Store Path Relative To)	ドメインモデルのグローバルなパス設定を参照します。デフォルト値は、JBoss Enterprise Application Platform 6 のデータディレクトリで、**jboss.server.data.dir** プロパティーの値です。デフォルトは、管理対象ドメインの場合は ***EAP_HOME*/domain/data/**、スタンドアロンサーバーインスタンスの場合は ***EAP_HOME*/standalone/data/** です。**path** TM 属性の値は、このパスに相対的です。空の文字列を使用して、デフォルト動作を無効にし、**path** 属性の値が絶対パスとして強制的に扱われるようにします。	**/profile=default/subsystem=transactions/:write-attribute(name=object-store-relative-to,value=jboss.server.data.dir)**
ソケットバインディング (Socket Binding)	ソケットベースのメカニズムを使用する場合に、トランザクションマネージャーの回復およびトランザクション識別子の生成に使用するソケットバインディングの名前を指定します。一意の識別子を生成する詳しい情報は、**process-id-socket-max-ports** を参照してください。ソケットバインディングは、管理コンソールの **Server** タブでサーバーグループごとに指定されます。	**/profile=default/subsystem=transactions/:write-attribute(name=socket-binding,value=txn-recovery-environment)**
ソケットバインディングのステータス (Status Socket Binding)	トランザクションステータスマネージャーで使用するソケットバインディングを指定します。	**/profile=default/subsystem=transactions/:write-attribute(name=status-socket-binding,value=txn-status-manager)**
リカバリーリスナー (Recovery Listener)	トランザクションリカバリのプロセスがネットワークソケットをリッスンするかどうかを指定します。デフォルトは **false** です。	**/profile=default/subsystem=transactions/:write-attribute(name=recovery-listener,value=false)**

以下は、高度なオプションで、修正は管理 CLI を使用することでのみ可能です。デフォルト設定の変更は注意して行ってください。詳細は Red Hat グローバルサポートサービスにお問い合わせください。

表10.14 高度な TM 設定オプション

オプション	説明	CLI コマンド

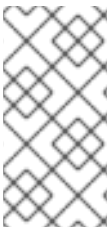
オプション	説明	CLI コマンド
jts	Java Transaction Service (JTS) トランザクションを使用するかどうかを指定します。デフォルトは **false** で、JTA トランザクションのみ使用します。	**/profile=default/subsystem=transactions/:write-attribute(name=jts,value=false)**
node-identifier	JTS サービスのノード識別子です。トランザクションマネージャーがリカバリ時にこれを使用するため、JTS サービスごとに一意でなければなりません。	**/profile=default/subsystem=transactions/:write-attribute(name=node-identifier,value=1)**
process-id-socket-max-ports	トランザクションマネージャーは、各トランザクションログに対し一意の識別子を作成します。一意の識別子を生成するメカニズムは 2 種類あります。ソケットベースのメカニズムとプロセスのプロセス識別子をベースにしたメカニズムです。 ソケットベースの識別子の場合、あるソケットを開くと、そのポート番号が識別子用に使用されます。ポートがすでに使用されている場合は、空きのポートが見つかるまで次のポートがプローブされます。**process-id-socket-max-ports** は、TM が失敗するまでに試行するソケットの最大値を意味します。デフォルト値は **10** です。	**/profile=default/subsystem=transactions/:write-attribute(name=process-id-socket-max-ports,value=10)**
process-id-uuid	**true** に設定すると、プロセス識別子を使用して各トランザクションに一意の識別子を作成します。そうでない場合は、ソケットベースのメカニズムが使用されます。デフォルトは **true** です。詳細は **process-id-socket-max-ports** を参照してください。	**/profile=default/subsystem=transactions/:write-attribute(name=process-id-uuid,value=true)**
use-hornetq-store	トランザクションログ用に、ファイルベースのストレージの代わりに HornetQ のジャーナルストレージメカニズムを使用します。デフォルトでは無効になっていますが、I/O パフォーマンスが向上します。別々のトランザクションマネージャーで JTS トランザクションを使用することは推奨されません。	**/profile=default/subsystem=transactions/:write-attribute(name=use-hornetq-store,value=false)**

バグを報告する

10.7.9. JTA トランザクションのエラー処理

10.7.9.1. トランザクションエラーの処理

トランザクションエラーは、多くの場合、タイミングに依存するため、解決するのが困難です。以下に、一部の一般的なエラーと、これらのエラーのトラブルシューティングに関するヒントを示します。



注記

これらのガイドラインはヒューリスティックエラーに適用されません。ヒューリスティックエラーが発生した場合は、「トランザクションにおけるヒューリスティックな結果の処理方法」 を参照し、Red Hat グローバルサポートサービスにお問い合わせください。

トランザクションがタイムアウトになったが、ビジネスロジックスレッドが認識しませんでした。

多くの場合、このようなエラーは、Hibernate がレイジーロードのためにデータベース接続を取得できない場合に発生します。頻繁に発生する場合は、タイムアウト値を大きくできます。「トランザクションマネージャーの設定」 を参照してください。

引き続き問題が解決されない場合は、パフォーマンスを向上させるために外部環境を調整するか、さらに効率的になるようコードを再構築できます。タイムアウトの問題が解消されない場合は、Red Hat グローバルサポートサービスにお問い合わせください。

トランザクションがすでにスレッドで実行されているか、`NotSupportedException` 例外が発生します。

`NotSupportedException` 例外は、通常、JTA トランザクションをネストしようとし、ネストがサポートされていないことを示します。トランザクションをネストしようとしないときは、多くの場合、スレッドプールタスクで別のトランザクションが開始されますが、トランザクションを中断または終了せずにタスクが終了します。

通常、アプリケーションは、これを自動的に処理する **`UserTransaction`** を使用します。その場合は、フレームワークに問題があることがあります。

コードで **`TransactionManager`** メソッドまたは **`Transaction`** メソッドを直接使用する場合は、トランザクションをコミットまたはロールバックするときに次の動作に注意してください。コードで **`TransactionManager`** メソッドを使用してトランザクションを制御する場合は、トランザクションをコミットまたはロールバックすると、現在のスレッドからトランザクションの関連付けが解除されます。ただし、コードで **`Transaction`** メソッドを使用する場合は、トランザクションを、実行中のスレッドに関連付けることができず、スレッドプールにスレッドを返す前にスレッドからトランザクションの関連付けを手動で解除する必要があります。

2 番目のローカルリソースを登録することはできません。

このエラーは、2 番目の非 XA リソースをトランザクションに登録しようとした場合に、発生します。1 つのトランザクションで複数のリソースが必要な場合、それらは XA である必要があります。

バグを報告する

10.8. ORB 設定

10.8.1. Common Object Request Broker Architecture (CORBA) について

*Common Object Request Broker Architecture (CORBA)*は、アプリケーションとサービスが複数の互換性

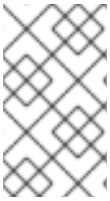
がない言語で記述され、異なるプラットフォームでホストされる場合でも、アプリケーションとサービスが連携することを可能にする標準です。CORBA 要求は *Object Request Broker (ORB)* というサーバーサイドコンポーネントにより JacORB コンポーネントを使用して処理されます。

ORB は *Java Transaction Service (JTS)* トランザクションに対して内部的に使用され、ユーザー独自のアプリケーションが使用することもできます。

バグを報告する

10.8.2. JTS トランザクション用 ORB の設定

JBoss Enterprise Application Platform のデフォルトインストールでは、ORB が無効になります。ORB は、コマンドライン管理 CLI を使用して有効にすることができます。



注記

管理対象ドメインでは、JacORB サブシステムが **full** および **full-ha** プロファイルでのみ利用可能です。スタンドアロンサーバーでは、**standalone-full.xml** または **standalone-full-ha.xml** 設定で利用可能です。

手順10.6 管理コンソールを使用した ORB の設定

1. **プロファイル設定を表示します。**
   管理コンソールの右上から **[Profiles]** (管理対象ドメイン) または **[Profile]** (スタンドアロンサーバー) を選択します。管理対象ドメインを使用する場合は、左上にある選択ボックスから **[full]** または **[full-ha]** プロファイルを選択します。

2. **Initializers 設定の変更**
   必要な場合は、左側にある **[Subsystems]** メニューを展開します。**[Container]** サブメニューを展開し、**[JacORB]** をクリックします。

   メイン画面に表示されるフォームで、**[Initializers]** タブを選択し、**[Edit]** ボタンをクリックします。

   [Security] の値を **on** に設定して、セキュリティーインターセプターを有効にします。

   JTS 用 ORB を有効にするには、**[Transaction Interceptors]** 値をデフォルトの **spec** ではなく **on** に設定します。

   これらの値に関する詳細な説明については、フォームの **[Need Help?]** リンクを参照してください。値の編集が完了したら、**[Save]** をクリックします。

3. **高度な ORB 設定**
   高度な設定オプションについては、フォームの他のセクションを参照してください。各セクションには、パラメーターに関する詳細な情報とともに **[Need Help?]** リンクが含まれます。

管理 CLI を使用して ORB を設定

管理 CLI を使用して ORB の各側面を設定できます。以下のコマンドは、管理コンソールに対するイニシャライザーに上記の手順と同じ値を設定します。これは、JTS と使用する ORB の最小設定です。

これらのコマンドは、**full** プロファイルを使用して管理対象ドメインに対して設定されます。必要な場合は、設定する必要があるプロファイルに合わせてプロファイルを変更します。スタンドアロンサーバーを使用する場合は、コマンドの **/profile=full** 部分を省略します。

例10.3 セキュリティーインターセプターの有効化

```
/profile=full/subsystem=jacorb/:write-attribute(name=security,value=on)
```

例10.4 JTS 用 ORB の有効化

```
/profile=full/subsystem=jacorb/:write-
attribute(name=transactions,value=on)
```

バグを報告する

10.9. トランザクションに関する参考資料

10.9.1. JBoss Transactions エラーと例外

UserTransaction クラスのメソッドがスローする例外に関する詳細は、http://download.oracle.com/javaee/1.3/api/javax/transaction/UserTransaction.html の『UserTransaction API』 の仕様を参照してください。

バグを報告する

10.9.2. JTA クラスタリングの制限事項

JTA トランザクションは、複数の JBoss Enterprise Application Platform インスタンスでクラスター化できません。そのため、JTS トランザクションを使用します。

JTS トランザクションを使用するには、ORB を設定する必要があります (「JTS トランザクション用 ORB の設定」)。

バグを報告する

10.9.3. JTA トランザクションの例

この例では、 JTA トランザクションを開始、コミット、およびロールバックする方法を示します。使用している環境に合わせて接続およびデータソースパラメーターを調整し、データベースで 2 つのテストテーブルをセットアップする必要があります。

例10.5 JTA トランザクションの例

```
public class JDBCExample {
    public static void main (String[] args) {
        Context ctx = new InitialContext();
        // Change these two lines to suit your environment.
        DataSource ds = (DataSource)ctx.lookup("jdbc/ExampleDS");
        Connection conn = ds.getConnection("testuser", "testpwd");
        Statement stmt = null; // Non-transactional statement
        Statement stmtx = null; // Transactional statement
        Properties dbProperties = new Properties();

        // Get a UserTransaction
```

```
        UserTransaction txn = new
InitialContext().lookup("java:comp/UserTransaction");

        try {
            stmt = conn.createStatement();  // non-tx statement

            // Check the database connection.
            try {
                stmt.executeUpdate("DROP TABLE test_table");
                stmt.executeUpdate("DROP TABLE test_table2");
            }
            catch (Exception e) {
                // assume not in database.
            }

            try {
                stmt.executeUpdate("CREATE TABLE test_table (a
INTEGER,b INTEGER)");
                stmt.executeUpdate("CREATE TABLE test_table2 (a
INTEGER,b INTEGER)");
            }
            catch (Exception e) {
            }

            try {
                System.out.println("Starting top-level transaction.");

                txn.begin();

                stmtx = conn.createStatement(); // will be a tx-
statement

                // First, we try to roll back changes

                System.out.println("\nAdding entries to table 1.");

                stmtx.executeUpdate("INSERT INTO test_table (a, b)
VALUES (1,2)");

                ResultSet res1 = null;

                System.out.println("\nInspecting table 1.");

                res1 = stmtx.executeQuery("SELECT * FROM test_table");

                while (res1.next()) {
                    System.out.println("Column 1: "+res1.getInt(1));
                    System.out.println("Column 2: "+res1.getInt(2));
                }
                System.out.println("\nAdding entries to table 2.");

                stmtx.executeUpdate("INSERT INTO test_table2 (a, b)
VALUES (3,4)");
                res1 = stmtx.executeQuery("SELECT * FROM test_table2");

                System.out.println("\nInspecting table 2.");
```

```
                while (res1.next()) {
                    System.out.println("Column 1: "+res1.getInt(1));
                    System.out.println("Column 2: "+res1.getInt(2));
                }

                System.out.print("\nNow attempting to rollback
changes.");

                txn.rollback();

                // Next, we try to commit changes
                txn.begin();
                stmtx = conn.createStatement();
                ResultSet res2 = null;

                System.out.println("\nNow checking state of table 1.");

                res2 = stmtx.executeQuery("SELECT * FROM test_table");

                while (res2.next()) {
                    System.out.println("Column 1: "+res2.getInt(1));
                    System.out.println("Column 2: "+res2.getInt(2));
                }

                System.out.println("\nNow checking state of table 2.");

                stmtx = conn.createStatement();

                res2 = stmtx.executeQuery("SELECT * FROM test_table2");

                while (res2.next()) {
                    System.out.println("Column 1: "+res2.getInt(1));
                    System.out.println("Column 2: "+res2.getInt(2));
                }

                txn.commit();
            }
            catch (Exception ex) {
                ex.printStackTrace();
                System.exit(0);
            }
        }
        catch (Exception sysEx) {
            sysEx.printStackTrace();
            System.exit(0);
        }
    }
}
```

バグを報告する

10.9.4. JBoss トランザクション JTA 向け API ドキュメンテーション

JBoss Enterprise Application Platform のトランザクションサブシステム向けAPI ドキュメンテーションは、以下の場所で取得できます。

- UserTransaction - http://download.oracle.com/javaee/1.3/api/javax/transaction/UserTransaction.html

JBoss Development Studio を使用してアプリケーションを開発する場合は、 API ドキュメンテーションが **[Help]** メニューに含まれています。

バグを報告する

第11章 HIBERNATE

11.1. HIBERNATE CORE について

Hibernate Core は、オブジェクト／関係マッピングライブラリです。これは、Java クラスをデータベーステーブルにマッピングするためのフレームワークを提供するため、アプリケーションがデータベースと対話しないことが可能になります。

詳細は 「Hibernate EntityManager」 および 「JPA について」 を参照してください。

バグを報告する

11.2. JAVA 永続 API (JPA)

11.2.1. JPA について

Java Persistence API (JPA) は、Java プロジェクトで永続性を利用する際の規格です。Java EE 6 アプリケーションは、**SecurityContext** に文書でまとめられている Java Persistence 2.0 仕様を利用します。

Hibernate EntityManager は、JPA 2.0 specification で定義されている、プログラミングインターフェースおよびライフサイクルルールを実装します。Hibernate EntityManager により、JBoss Enterprise Application Platform にて完全な Java Persistence ソリューションを得ることができます。

JBoss Enterprise Application Platform 6 は Java Persistence 2.0 仕様に完全準拠しています。Hibernate はこの仕様に対しさらに機能を提供しています。

JPA および JBoss Enterprise Application Platform 6 を利用するには、**bean-validation**、**greeter**、**kitchensink** クイックスタート: 「Java EE クイックスタートサンプルへのアクセス」 を参照してくだい。

バグを報告する

11.2.2. Hibernate EntityManager

Hibernate EntityManager は、JPA 2.0 specificationで定義されている、プログラミングインターフェースおよびライフサイクルルールを実装します。Hibernate EntityManager により、JBoss Enterprise Application Platform にて完全な Java Persistence ソリューションを得ることができます。

Java Persistence あるいは Hibernate に関する詳細情報は、 「JPA について」 および 「Hibernate Core について」 を参照してください。

バグを報告する

11.2.3. 使用開始

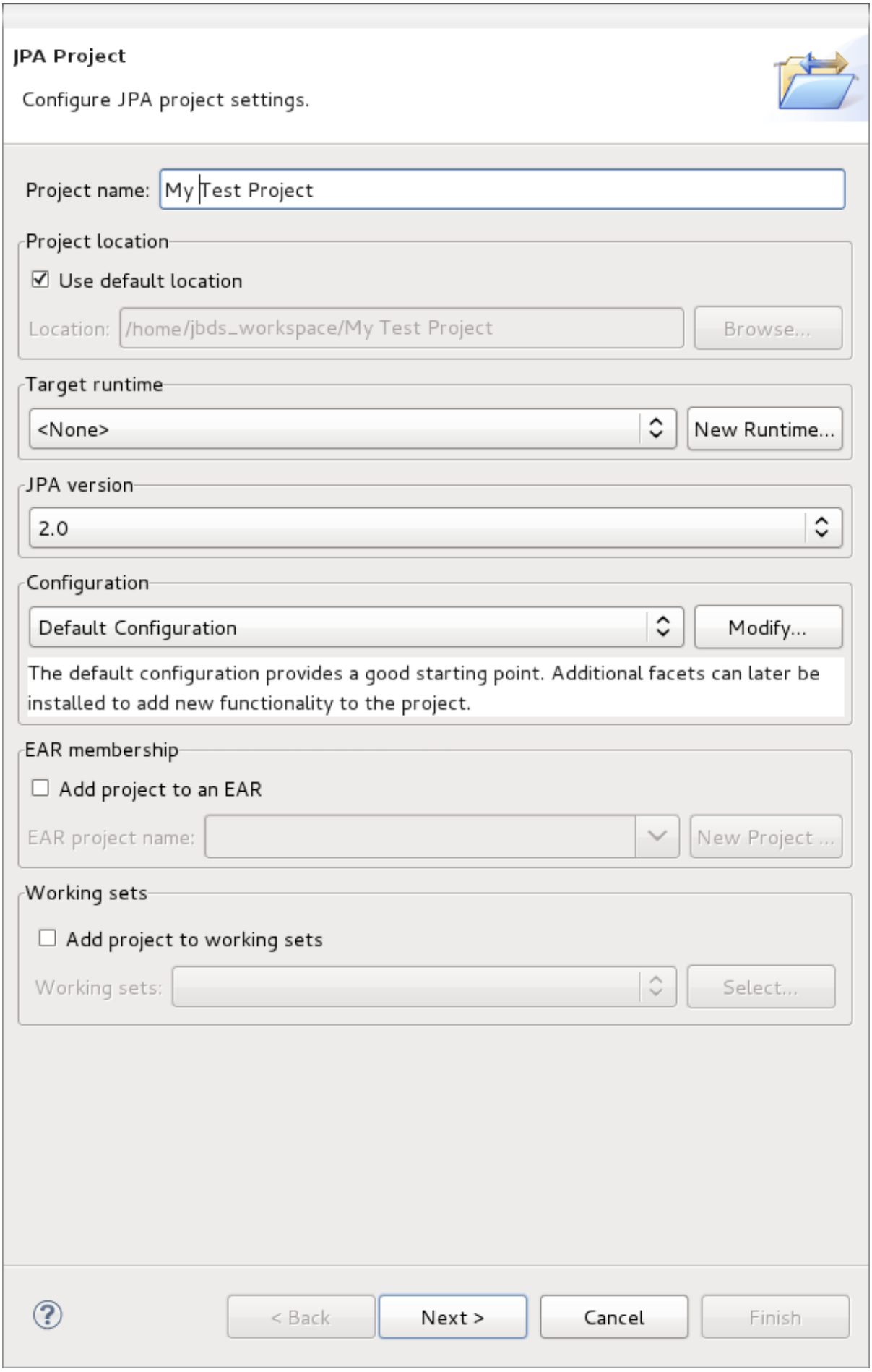
11.2.3.1. JBoss Developer Studio における JPA プロジェクトの作成

概要

この例では、JBoss Developer Studio で JPA プロジェクトを作成するために必要な手順について取り上げます。

手順11.1 JBoss Developer Studio における JPA プロジェクトの作成

1. JBoss Developer Studio のウインドウで **[File]** → **[New]** → **[JPA Project]** と選択します。
2. プロジェクトダイアログにプロジェクト名を入力します。

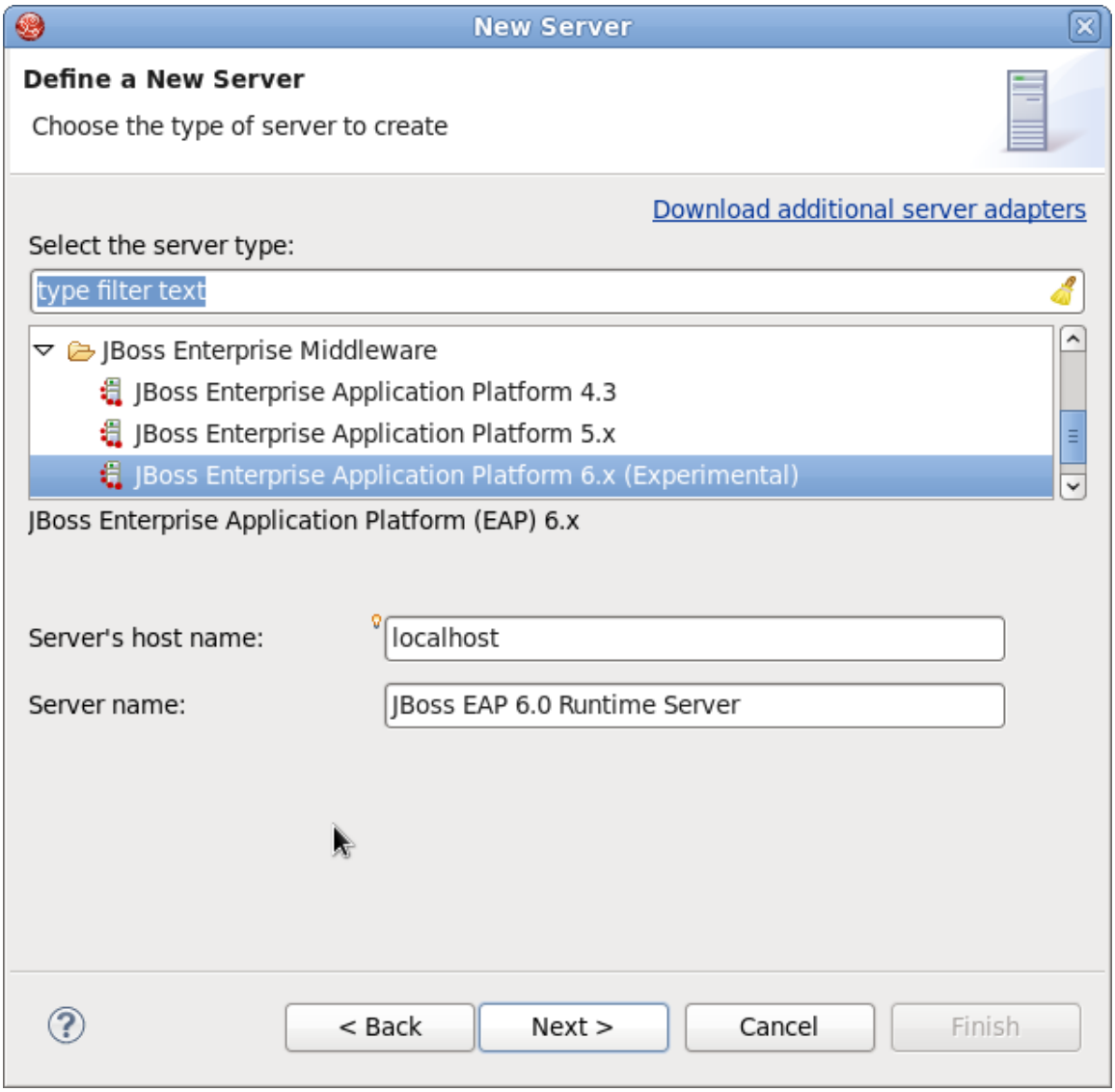


3. ドロップダウンボックスよりターゲットランタイムを選択します。

4. a. ターゲットランタイムがない場合は **[Target Runtime]** をクリックします。

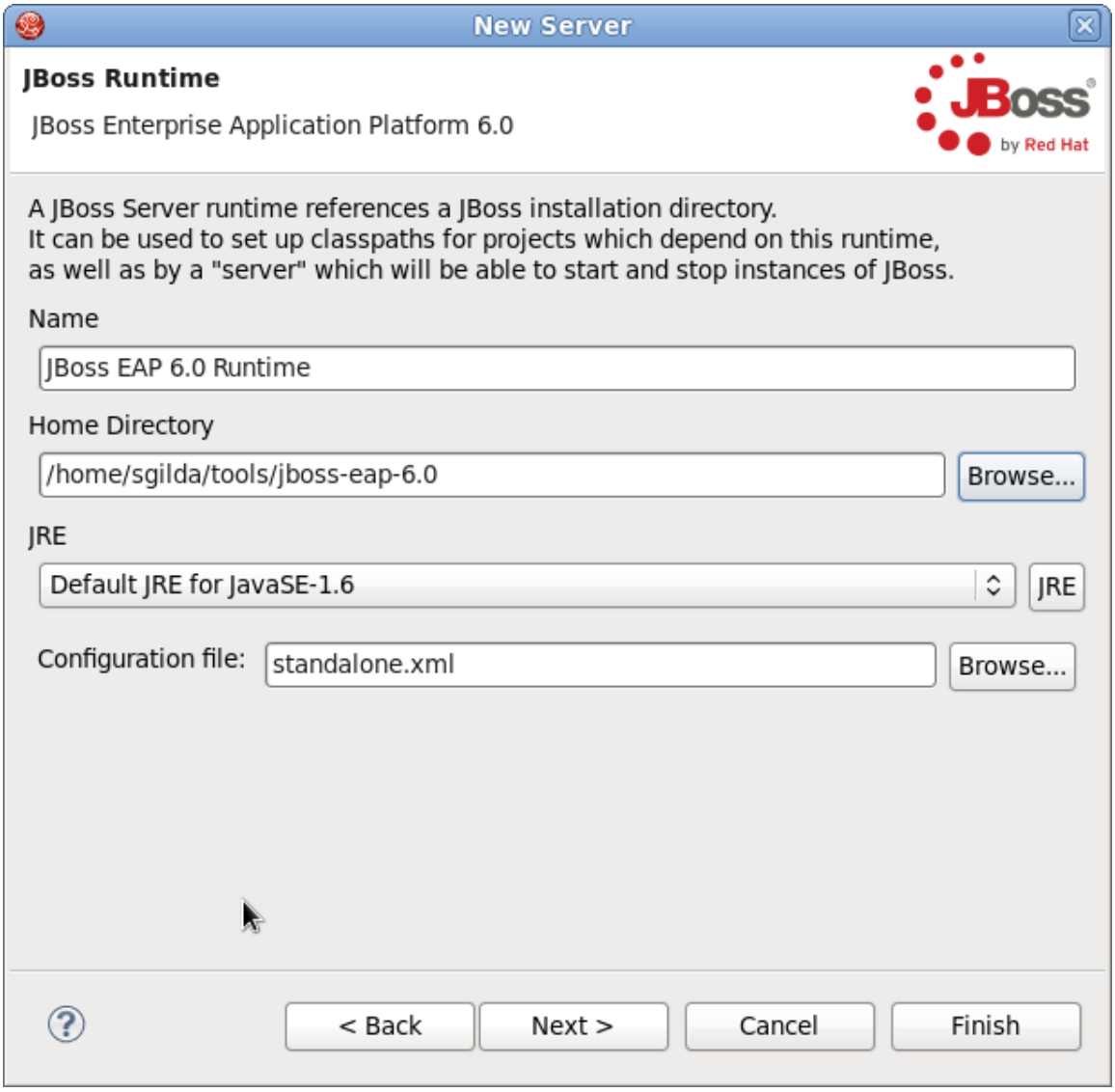
b. リストで JBoss Community Folder を探します。

c. JBoss Enterprise Application Platform 6.x ランタイムの選択



d. **[Next]** をクリックします。

e. Home Directory フィールドで **[Browse]** をクリックし、JBoss EAP ソースフォルダーを Home Directory として設定します。



f. **[Finish]** をクリックします。

5. **[Next]** をクリックします。
6. ビルドパスウインドウのソースフォルダーはデフォルトのままにし、**[Next]** をクリックします。
7. Platform ドロップダウンで必ず Hibernate (JPA 2.x) が選択されているようにしてください。
8. **[Finish]** をクリックします。
9. 要求されたら、JPA パースペクティブウインドウを開くかどうかを選択します。

バグを報告する

11.2.3.2. JBoss Developer Studio での永続設定ファイルの作成

概要

このトピックでは、JBoss Developer Studio を使用して Java プロジェクトで **persistence.xml** ファイルを作成するプロセスについて説明します。

要件

- 「JBoss Developer Studio の起動」

手順11.2 新しい永続性設定ファイルを作成および設定する

1. JBoss Developer Studio で EJB 3.x プロジェクトを開きます。
2. **[Project Explorer]** パネルのプロジェクトのルートディレクトリーを右クリックします。
3. **[New]** → **[Other...]** を選択します。
4. **XML** フォルダーから **[XML File]** を選択し、**[Next]** をクリックします。
5. 親ディレクトリとして **ejbModule/META-INF** フォルダーを選択します。
6. **persistence.xml** ファイルに名前を付け、**[Next]** をクリックします。
7. **[Create XML file from an XML schema file]** を選択し、**[Next]** をクリックします。
8. **[Select XML Catalog entry]** リストから **http://java.sun.com/xml/ns/persistence/persistence_2.0.xsd** を選択し、**[Next]** をクリックします。

9. **[Finish]** をクリックし、ファイルを作成します。

結果

persistence.xml が **META-INF/** フォルダーに作成され、設定できる状態になります。サンプルファイルは、「永続設定ファイルの例」で取得できます。

バグを報告する

11.2.3.3. 永続設定ファイルの例

例11.1 persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
   version="2.0">
   <persistence-unit name="example" transaction-type="JTA">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-
source>
      <mapping-file>ormap.xml</mapping-file>
      <jar-file>TestApp.jar</jar-file>
      <class>org.test.Test</class>
      <shared-cache-mode>NONE</shared-cache-mode>
      <validation-mode>CALLBACK</validation-mode>
      <properties>
         <property name="hibernate.dialect"
value="org.hibernate.dialect.H2Dialect"/>
```

```
            <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
        </properties>
    </persistence-unit>
</persistence>
```

バグを報告する

11.2.3.4. JBoss Developer Studio の Hibernate 設定ファイルの作成

要件

- 「JBoss Developer Studio の起動」

概要

本トピックでは、JBoss Developer Studio を使用して Java プロジェクトに **hibernate.cfg.xml** ファイルを作成するプロセスについて説明します。

手順11.3 Create a New Hibernate Configuration File

1. JBoss Developer Studio で Java プロジェクトを開きます。
2. **[Project Explorer]** パネルのプロジェクトのルートディレクトリーを右クリックします。
3. **[New]** → **[Other...]** を選択します。
4. **Hibernate** フォルダーから **[Hibernate Configuration File]** を選択し、**[Next]** をクリックします。
5. **src/** ディレクトリを選択し、**[Next]** をクリックします。
6. 以下を設定します。
   - セッションファクトリー名
   - データベースの方言
   - ドライバークラス
   - 接続 URL
   - ユーザー名
   - パスワード
7. **[Finish]** をクリックし、ファイルを作成します。

結果

src/ フォルダーに **hibernate.cfg.xml** が作成されます。ファイル例は「Hibernate 設定ファイルの例」 を参照してください。

バグを報告する

11.2.3.5. Hibernate 設定ファイルの例

例11.2 hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Datasource Name -->
        <property name="connection.datasource">ExampleDS</property>

        <!-- SQL dialect -->
        <property
name="dialect">org.hibernate.dialect.H2Dialect</property>

        <!-- Enable Hibernate's automatic session context management --
>
        <property
name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache  -->
        <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</propert
y>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">update</property>

        <mapping
resource="org/hibernate/tutorial/domain/Event.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

バグを報告する

11.2.4. 設定

11.2.4.1. Hibernate 設定プロパティー

表11.1 プロパティ

プロパティー名	説明
hibernate.dialect	Hibernate の **org.hibernate.dialect.Dialect** のクラス名。Hibernate で、特定のリレーショナルデータベースに最適化された SQL を生成できるようになります。 ほとんどのケースで、Hibernate は、JDBC ドライバーにより返された **JDBC メタデータ**に基づいて正しい **org.hibernate.dialect.Dialect** 実装を選択できます。
hibernate.show_sql	ブール変数。SQL ステートメントをすべてコンソールに書き込みます。これは、ログカテゴリー **org.hibernate.SQL** を **debug** に設定することと同じです。
hibernate.format_sql	ブール変数。SQL をログとコンソールにプリティプリントします。
hibernate.default_schema	修飾されていないテーブル名を、生成された SQL の該当するスキーマ/テーブルスペースで修飾します。
hibernate.default_catalog	修飾されていないテーブル名を、生成された SQL の該当するカタログで修飾します。
hibernate.session_factory_name	**org.hibernate.SessionFactory** が、作成後に JNDI のこの名前に自動的にバインドされます。たとえば、**jndi/composite/name** のようになります。
hibernate.max_fetch_depth	シングルエンドの関連 (1 対 1 や多対 1 など) に対して外部結合フェッチツリーの最大の「深さ」を設定します。**0** を設定するとデフォルトの外部結合フェッチが無効になります。推奨値は、**0～3** です。
hibernate.default_batch_fetch_size	関連付けの Hibernate 一括フェッチに対するデフォルトサイズを設定します。推奨値は、**4**、**8**、および **16** です。
hibernate.default_entity_mode	この **SessionFactory** から開かれたすべてのセッションに対するエンティティー表現のデフォルトモードを設定します。値には **dynamic-map**、**dom4j**、**pojo** などがあります。
hibernate.order_updates	ブール変数。Hibernate で、更新されるアイテムの主キー値で SQL 更新の順番付けを行います。これにより、高度な並列システムにおけるトランザクションデッドロックが軽減されます。
hibernate.generate_statistics	ブール変数。有効にすると、Hibernate がパフォーマンスのチューニングに役に立つ統計情報を収集します。

プロパティー名	説明
hibernate.use_identifier_rollback	ブール変数。有効にすると、オブジェクトが削除されたときに、生成された識別子プロパティーがデフォルト値にリセットされます。
hibernate.use_sql_comments	ブール変数。有効にすると、デバッグを簡単にするために Hibernate が SQL 内にコメントを生成します。デフォルト値は **false** です。
hibernate.id.new_generator_mappings	ブール変数。**@GeneratedValue** を使用する場合に関係するプロパティーです。新しい **IdentifierGenerator** 実装が **javax.persistence.GenerationType.AUTO**、**javax.persistence.GenerationType.TABLE**、または **javax.persistence.GenerationType.SEQUENCE** に対して使用されるかどうかを示します。後方互換性を維持するために、デフォルト値は **false** になっています。

重要

@GeneratedValue を使用する新しいプロジェクトは **hibernate.id.new_generator_mappings=true** も設定することが推奨されます。これは、新しいジェネレーターはより効率的で JPA 2 仕様のセマンティックに近いためです。

しかし、この設定は既存データベースとの後方互換性を維持しません (シーケンスやテーブルが ID 生成に使用される場合)。

バグを報告する

11.2.4.2. Hibernate JDBC と接続プロパティー

表11.2 プロパティ

プロパティー名	説明
hibernate.jdbc.fetch_size	JDBC のフェッチサイズを判断するゼロでない値です (**Statement.setFetchSize()** を呼び出します)。
hibernate.jdbc.batch_size	Hibernate による JDBC2 バッチ更新の使用を有効にするゼロでない値です。推奨値は、**5〜30** です。

プロパティー名	説明
hibernate.jdbc.batch_versioned_data	ブール変数。JDBC ドライバーが **`executeBatch()`** から正しい行数を返す場合は、このプロパティーを **`true`** に設定します。Hibernate は自動的にバージョン化されたデータにバッチ処理された DML を使用します。デフォルト値は **`false`** です。
hibernate.jdbc.factory_class	カスタム **`org.hibernate.jdbc.Batcher`** を選択します。ほとんどのアプリケーションにはこの設定プロパティーは必要ありません。
hibernate.jdbc.use_scrollable_resultset	ブール変数。Hibernate による JDBC2 のスクロール可能な結果セットの使用を有効にします。このプロパティーはユーザーが提供した JDBC 接続を使用する場合にのみ必要です。その他の場合、Hibernate は接続メタデータを使用します。
hibernate.jdbc.use_streams_for_binary	ブール変数。システムレベルのプロパティーです。**`binary`** または **`serializable`** 型を JDBC へ読み書きしたり、JDBC から読み書きしたりする場合にストリームを使用します。
hibernate.jdbc.use_get_generated_keys	ブール変数。JDBC3 **`PreparedStatement.getGeneratedKeys()`** を使用して、挿入後にネイティブで生成された鍵を取得できるようにします。JDBC3+ ドライバーと JRE1.4+ が必要です。JDBC ドライバーに Hibernate 識別子ジェネレーターの問題がある場合は false に設定します。デフォルトでは、接続メタデータを使用してドライバーの機能を判断しようとします。
hibernate.connection.provider_class	JDBC 接続を Hibernate に提供するカスタム **`org.hibernate.connection.ConnectionProvider`** のクラス名です。
hibernate.connection.isolation	JDBC トランザクションの分離レベルを設定します。**`java.sql.Connection`** で意味のある値をチェックしますが、ほとんどのデータベースはすべての分離レベルをサポートするとは限らず、一部のデータベースは標準的でない分離を追加的に定義します。標準的な値は **`1, 2, 4, 8`** です。
hibernate.connection.autocommit	ブール変数。このプロパティーの使用は推奨されません。JDBC でプールされた接続に対して自動コミットを有効にします。

プロパティー名	説明
hibernate.connection.release_mode	Hibernate が JDBC 接続を開放するタイミングを指定します。デフォルトでは、セッションが明示的に閉じられるか切断されるまで JDBC 接続が保持されます。デフォルト値である **auto** では、JTA および CMT トランザクションストラテジーに対して **after_statement** が選択され、JDBC トランザクションストラテジーに対して **after_transaction** が選択されます。 使用可能な値は、**auto** (デフォルト値) **|on_close |after_transaction|after_statement** です。 この設定により、**SessionFactory.openSession** から返された**セッション**のみが影響を受けます。**SessionFactory.getCurrentSession** から取得された**セッション**の場合、使用のために設定された **CurrentSessionContext** 実装はこれらの**セッション**の接続リリースモードを制御します。
hibernate.connection.**<propertyName>**	JDBC プロパティー **<propertyName>** を **DriverManager.getConnection()** に渡します。
hibernate.jndi.**<propertyName>**	プロパティー **<propertyName>** を JNDI **InitialContextFactory** に渡します。

バグを報告する

11.2.4.3. Hibernate キャッシュプロパティー

表11.3 プロパティー

プロパティー名	説明
hibernate.cache.provider_class	カスタム **CacheProvider** のクラス名。
hibernate.cache.use_minimal_puts	ブール変数です。2 次キャッシュの操作を最適化し、読み取りを増やして書き込みを最小限にします。これはクラスター化されたキャッシュで最も便利な設定であり、Hibernate 3 ではクラスター化されたキャッシュの実装に対してデフォルトで有効になっています。
hibernate.cache.use_query_cache	ブール変数です。クエリーキャッシュを有効にします。各クエリーをキャッシュ可能に設定する必要があります。

プロパティー名	説明
hibernate.cache.use_second_level_cache	ブール変数です。**<cache>** マッピングを指定するクラスに対してデフォルトで有効になっている 2 次キャッシュを完全に無効にするため使用されます。
hibernate.cache.query_cache_factory	カスタム **QueryCache** インターフェースのクラス名です。デフォルト値は組み込みの **StandardQueryCache** です。
hibernate.cache.region_prefix	2 次キャッシュのリージョン名に使用するプレフィックスです。
hibernate.cache.use_structured_entries	ブール変数です。人間が解読可能な形式でデータを 2 次キャッシュに保存するよう Hibernate を設定します。
hibernate.cache.default_cache_concurrency_strategy	**@Cacheable** または **@Cache** が使用される場合に使用するデフォルトの **org.hibernate.annotations.CacheConcurrencyStrategy** の名前を付与するため使用される設定です。**@Cache(strategy="..")** を使用してこのデフォルト値が上書きされます。

バグを報告する

11.2.4.4. Hibernate トランザクションプロパティー

表11.4 プロパティ

プロパティー名	説明
hibernate.transaction.factory_class	Hibernate **Transaction** API と使用する **TransactionFactory** のクラス名です。デフォルト値は **JDBCTransactionFactory**) です。
jta.UserTransaction	アプリケーションサーバーから JTA **UserTransaction** を取得するために **JTATransactionFactory** により使用される JNDI 名。
hibernate.transaction.manager_lookup_class	**TransactionManagerLookup** のクラス名。JVM レベルのキャッシングが有効になっている場合や、JTA 環境の hilo ジェネレーターを使用する場合に必要です。
hibernate.transaction.flush_before_completion	ブール変数。有効な場合、トランザクションの完了前フェーズの間にセッションが自動的にフラッシュされます。ビルトインおよび自動セッションコンテキスト管理が推奨されます。

プロパティー名	説明
hibernate.transaction.auto_close_session	ブール変数。有効な場合、トランザクションの完了後フェーズの間にセッションが自動的に閉じられます。ビルトインおよび自動セッションコンテキスト管理が推奨されます。

バグを報告する

11.2.4.5. その他の Hibernate プロパティー

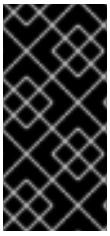
表11.5 プロパティ

プロパティー名	説明
hibernate.current_session_context_class	「現在」の **Session** のスコープに対するカスタムストラテジーを提供します。値には **jta|thread|managed|custom.Class** があります。
hibernate.query.factory_class	**org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory** または **org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory** の HQL パーサー実装を選択します。
hibernate.query.substitutions	Hibernate クエリーのトークンと SQL トークンとのマッピングに使用します (トークンは関数名またはリテラル名である場合があります)。たとえば、**hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC** のようになります。
hibernate.hbm2ddl.auto	**SessionFactory** が作成されると、スキーマ DDL を自動的に検証し、データベースにエクスポートします。**create-drop** を使用すると、**SessionFactory** が明示的に閉じられたときにデータベーススキーマが破棄されます。プロパティー値のオプションは、**validate|update|create|create-drop** になります。
hibernate.hbm2ddl.import_files	**SessionFactory** 作成中に実行される SQL DML ステートメントが含まれる任意ファイルの名前 (コンマ区切り)。テストやデモに便利です。たとえば、INSERT ステートメントを追加すると、デプロイ時に最小限のデータセットがデータベースに入力されます。例としては、**/humans.sql,/dogs.sql** のようになります。 特定ファイルのステートメントは後続ファイルのステートメントの前に実行されるため、ファイルの順番に注意する必要があります。これらのステートメントはスキーマが作成された場合のみ実行されます (**hibernate.hbm2ddl.auto** が **create** または **create-drop** に設定された場合など)。

プロパティー名	説明
hibernate.hbm2ddl.import_files_sql_extractor	カスタム **ImportSqlCommandExtractor** のクラス名。デフォルト値は組み込みの **SingleLineSqlCommandExtractor** です。各インポートファイルから単一の SQL ステートメントを抽出する専用のパーサーを実装する時に便利です。Hibernate は、複数行にまたがる命令/コメントおよび引用符で囲まれた文字列をサポートする **MultipleLinesSqlCommandExtractor** も提供します (各ステートメントの最後にセミコロンが必要です)。
hibernate.bytecode.use_reflection_optimizer	ブール変数。**hibernate.cfg.xml** ファイルで設定できないシステムレベルのプロパティーです。ランタイムリフレクションの代わりにバイトコード操作の使用を有効にします。リフレクションは、トラブルシューティングを行うときに便利な場合があります。オプティマイザーが無効の場合でも Hibernate には CGLIB または javassist が常に必要です。
hibernate.bytecode.provider	javassist または cglib をバイト操作エンジンとして使用することができます。デフォルトでは **javassist** が使用されます。プロパティー値は **javassist** または **cglib** のいずれかです。

バグを報告する

11.2.4.6. Hibernate SQL 方言



重要

hibernate.dialect プロパティーをアプリケーションデータベースの適切な **org.hibernate.dialect.Dialect** サブクラスに設定する必要があります。方言が指定されている場合、Hibernate は他のプロパティーの一部に実用的なデフォルトを使用します。そのため、これらのプロパティーを手作業で指定する必要はありません。

表11.6 SQL 方言 (hibernate.dialect)

RDBMS	方言
DB2	**org.hibernate.dialect.DB2Dialect**
DB2 AS/400	**org.hibernate.dialect.DB2400Dialect**
DB2 OS390	**org.hibernate.dialect.DB2390Dialect**
PostgreSQL	**org.hibernate.dialect.PostgreSQLDialect**
MySQL5	**org.hibernate.dialect.MySQL5Dialect**

RDBMS	方言
InnoDB を用いる MYSQL5	`org.hibernate.dialect.MySQL5InnoDBDialect`
MyISAM を用いる MySQL	`org.hibernate.dialect.MySQLMyISAMDialect`
Oracle (全バージョン)	`org.hibernate.dialect.OracleDialect`
Oracle 9i	`org.hibernate.dialect.Oracle9iDialect`
Oracle 10g	`org.hibernate.dialect.Oracle10gDialect`
Oracle 11g	`org.hibernate.dialect.Oracle10gDialect`
Sybase	`org.hibernate.dialect.SybaseASE15Dialect`
Sybase Anywhere	`org.hibernate.dialect.SybaseAnywhereDialect`
Microsoft SQL Server 2000	`org.hibernate.dialect.SQLServerDialect`
Microsoft SQL Server 2005	`org.hibernate.dialect.SQLServer2005Dialect`
Microsoft SQL Server 2008	`org.hibernate.dialect.SQLServer2008Dialect`
SAP DB	`org.hibernate.dialect.SAPDBDialect`
Informix	`org.hibernate.dialect.InformixDialect`
HypersonicSQL	`org.hibernate.dialect.HSQLDialect`
H2 Database	`org.hibernate.dialect.H2Dialect`
Ingres	`org.hibernate.dialect.IngresDialect`
Progress	`org.hibernate.dialect.ProgressDialect`

RDBMS	方言
Mckoi SQL	`org.hibernate.dialect.MckoiDialect`
Interbase	`org.hibernate.dialect.InterbaseDialect`
Pointbase	`org.hibernate.dialect.PointbaseDialect`
FrontBase	`org.hibernate.dialect.FrontbaseDialect`
Firebird	`org.hibernate.dialect.FirebirdDialect`

バグを報告する

11.2.5. 2 次キャッシュ

11.2.5.1. 2 次キャッシュについて

2 次キャッシュとは、アプリケーションセッション以外で永続的に情報を保持するローカルのデータストアのことです。このキャッシュは永続プロバイダーにより管理されており、アプリケーションとデータを分けることでランタイム効率の改善をはかることができます。

JBoss Enterprise Application Platform 6 は以下を目的としたキャッシングをサポートします。

- Web セッションのクラスタリング
- ステートフルセッション Bean のクラスタリング
- SSO クラスタリング
- Hibernate 2 次キャッシュ

各キャッシュコンテナは「repl」と「dist」キャッシュを定義します。これらのキャッシュは、ユーザーアプリケーションで直接使用しないでください。

バグを報告する

11.2.5.2. Hibernate 用 2 次キャッシュを設定する

このトピックでは、Hibernate の 2 次レベルキャッシュとして動作するよう Infinispan を有効にする場合の設定要件について説明します。

手順11.4 `hibernate.cfg.xml` ファイルを作成および編集する

1. **hibernate.cfg.xml ファイルを作成します。**
   デプロイメントのクラスパスで`hibernate.cfg.xml` を作成します。詳細については、「JBoss Developer Studio の Hibernate 設定ファイルの作成」を参照してください。

2. XML の次の行をアプリケーションの **hibernate.cfg.xml** ファイルに追加します。この XML は **<session-factory>** タグ内部にある必要があります。

```
<property
name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
```

3. 以下のいずれかを **hibernate.cfg.xml** ファイルの **<session-factory>** セクションに追加します。

   - **Infinispan CacheManager が JNDI にバインドされる場合：**

```
<property name="hibernate.cache.region.factory_class">
    org.hibernate.cache.infinispan.JndiInfinispanRegionFactory
</property>
<property name="hibernate.cache.infinispan.cachemanager">
    java:CacheManager
</property>
```

   - **Infinispan CacheManager がスタンドアロンである場合：**

```
<property name="hibernate.cache.region.factory_class">
    org.hibernate.cache.infinispan.InfinispanRegionFactory
</property>
```

結果

Infinispan が Hibernate の 2 次レベルキャッシュとして設定されます。

バグを報告する

11.3. HIBERNATE アノテーション

11.3.1. Hibernate アノテーション

表11.7 Hibernate によって定義されるアノテーション

アノテーション	説明
AccessType	プロパティーのアクセスタイプ。
Any	複数のエンティティータイプを示す ToOne 関連を定義します。 メタデータ弁別子カラムより according エンティティータイプを一致します。このようなマッピングは最低限にするべきです。
AnyMetaDef	@Any および @manyToAny メタデータを定義します。
AnyMedaDefs	@Any および @ManyToAny のメタデータセットを定義します。エンティティーレベルまたはパッケージレベルで定義が可能です。

アノテーション	説明
BatchSize	SQL ローディングのバッチサイズ。
キャッシュ	ルートエンティティーまたはコレクションにキャッシングストラテジーを追加します。
Cascade	関連付けにカスケードストラテジーを適用します。
Check	クラス、プロパティー、コレクションのいずれかのレベルで定義できる任意の SQL チェック制約です。
Columns	カラムのアレイをサポートします。コンポーネントユーザータイプのマッピングに便利です。
ColumnTransformer	カラムからの値の読み取りやカラムへの値の書き込みに使用されるカスタム SQL 表現です。直接的なオブジェクトのロードや保存、クエリに使用されます。write 表現には必ず値に対して 1 つの「?」プレースホルダーが含まれなければなりません。
ColumnTransformers	@ColumnTransformer の複数アノテーションです。複数のカラムがこの挙動を使用する場合に便利です。
DiscriminatorFormula	ルートエントリーに置かれる弁別子の公式です。
DiscriminatorOptions	Hibernate 固有の弁別子プロパティーを表現する任意のアノテーションです。
Entity	Hibernate の機能でエンティティーを拡張します。
Fetch	特定の関連に使用されるフェッチングストラテジーを定義します。
FetchProfile	フェッチングストラテジープロファイルを定義します。
FetchProfiles	@FetchProfile の複数アノテーション。
フィルター	エンティティーまたはコレクションのターゲットエンティティーにフィルターを追加します。
FilterDef	フィルター定義。
FilterDefs	フィルター定義のアレイ。
FilterJoinTable	結合テーブルのコレクションへフィルターを追加します。

アノテーション	説明
FilterJoinTables	複数の @FilterJoinTable をコレクションへ追加します。
Filters	複数の @Filters を追加します。
数式	ほとんどの場所で @Column の代替として使用されます。公式は有効な SQL フラグメントである必要があります。
Generated	このアノテーション付けされたプロパティーはデータベースによって生成されます。
GenericGenerator	Hibernate ジェネレーターをデタイプを用いて記述するジェネレーターアノテーションです。
GenericGenerators	汎用ジェネレーター定義のアレイ。
Immutable	エンティティーまたはコレクションを不変としてマーク付けします。アノテーションがない場合、要素は可変となります。 不変のエンティティーはアプリケーションによって更新されないことがあります。不変エンティティーへの更新は無視されますが、例外はスローされません。 @Immutable をコレクションに付けるとコレクションは不変になるため、コレクションからの追加や削除およびコレクションへの追加や削除は許可されません。この結果、 HibernateException がスローされます。
Index	データベースのインデックスを定義します。
JoinFormula	ほとんどの場所で @JoinColumn の代替として使用されます。公式は有効な SQL フラグメントである必要があります。
LazyCollection	コレクションのレイジー状態を定義します。
LazyToOne	ToOne 関連のレイジー状態を定義します (OneToOne や ManyToOne など)。
Loader	Hibernate のデフォルトである FIND メソッドを上書きします。
ManyToAny	異なるエンティティー型を示す ToMany 関連を定義します。 メタデータ弁別子カラムより according エンティティータイプを一致します。このようなマッピングは最低限にするべきです。

アノテーション	説明
MapKeyType	永続マップのキータイプを定義します。
MetaValue	特定のエンティティータイプへ関連付けられる弁別子の値を表します。
NamedNativeQueries	Hibernate NamedNativeQuery オブジェクトを保持するよう NamedNativeQueries を拡張します。
NamedNativeQuery	Hibernate の機能で NamedNativeQuery を拡張します。
NamedQueries	Hibernate NamedQuery オブジェクトを保持するよう NamedQuery を拡張します。
NamedQuery	Hibernate の機能で NamedQuery を拡張します。
NaturalId	プロパティーがエンティティーのナチュラル ID の一部であることを指定します。
NotFound	関連上で要素が見つからなかった時に実行するアクションです。
OnDelete	コレクションやアレイ、結合されたサブクラスの削除に使用されるストラテジーです。OnDelete の 2 次テーブルはサポートされていません。
OptimisticLock	アノテーション付けされたプロパティーの変更によってエンティティーのバージョン番号が増加するかどうか。アノテーション付けされていない場合、プロパティーは楽観的ロックストラテジー (デフォルト) に関与します。
OptimisticLocking	エンティティーに適用される楽観的ロックのスタイルを定義するため使用されます。階層ではルートエンティティーのみに有効です。
OrderBy	SQL の順序付け (HQL の順序付けではない) を使用してコレクションの順序を付けます。
ParamDef	パラメーターの定義。
パラメーター	キーと値のパターン。
Parent	所有者 (通常は所有するエンティティー) へのポインターとしてプロパティーを参照します。
Persister	カスタムパーシスターを指定します。

アノテーション	説明
Polymorphism	Hibernate がエンティティーの階層に適用する多様性タイプを定義するため使用されます。
Proxy	特定クラスのレイジーおよびプロキシ設定。
RowId	Hibernate の ROWID マッピング機能をサポートします。
Sort	コレクションのソート (Java レベルのソート)。
接続元	バージョンおよびタイムスタンプバージョンプロパティーと併用するのに最適なアノテーションです。アノテーション値はタイムスタンプが生成される場所を決定します。
SQLDelete	Hibernate のデフォルトである DELETE メソッドを上書きします。
SQLDeleteAll	Hibernate のデフォルトである DELETE ALL メソッドを上書きします。
SQLInsert	Hibernate のデフォルトである INSERT INFO メソッドを上書きします。
SQLUpdate	Hibernate のデフォルトである UPDATE メソッドを上書きします。
Subselect	不変の読み取り専用エンティティーを指定のサブセレクト表現へマッピングします。
Synchronize	自動フラッシュが適切に行われ、派生したエンティティーに対するクエリが陳腐データを返さないようにします。ほとんどの場合でサブセレクトと共に使用されます。
Table	1 次または 2 次テーブルへの補足情報。
Tables	Table の複数アノテーション。
ターゲット	明示的なターゲットを定義し、リフレクションやジェネリクスで解決しないようにします。
Tuplizer	1 つのエンティティーまたはコンポーネントに対して単一の tuplizer を定義します。
Tuplizers	1 つのエンティティーまたはコンポーネントに対して tuplizer のセットを定義します。

アノテーション	説明
タイプ	Hibernate のタイプ。
TypeDef	Hibernate タイプの定義。
TypeDefs	Hibernate タイプ定義のアレイ。
Where	要素エンティティーまたはコレクションのターゲットエンティティーへ追加する where 節。この節は SQL で書かれます。
WhereJoinTable	コレクション結合テーブルへ追加する where 節。この節は SQL で書かれます。

バグを報告する

11.4. HIBERNATE クエリ言語

11.4.1. Hibernate クエリ言語

Hibernate クエリ言語 (HQL) と Java 永続クエリ言語 (JPQL) は、本質的に SQL と似ているオブジェクトモデルを重視したクエリ言語です。HQL は JPQL のスーパーセットです。HQL クエリは有効な JPQL クエリでないこともありますが、JPQL クエリは常に有効な HQL クエリになります。

HQL と JPQL は共にタイプセーフでないクエリ操作を実行します。基準 (criteria) クエリがタイプセーフなクエリを提供します。

バグを報告する

11.4.2. HQL ステートメント

HQL は **SELECT**、 **UPDATE**、 **DELETE**、および **INSERT** ステートメントを許可します。JPQL には HQL の **INSERT** ステートメントに相当するステートメントはありません。

重要

UPDATE または **DELETE** ステートメントを実行する場合は注意してください。

表11.8 HQL ステートメント

ステートメント	説明

ステートメント	説明
SELECT	HQL の **SELECT** ステートメントの BNF は次の通りです。 `select_statement :: = [select_clause] from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]` 最も簡単な HQL の **SELECT** ステートメントは次のような形式になります。 `from com.acme.Cat`
UDPATE	HQL の UPDATE ステートメントの BNF は JPQL と同じです。
DELETE	HQL の DELETE ステートメントの BNF は JPQL と同じです。

バグを報告する

11.4.3. INSERT ステートメントについて

HQL は **INSERT** ステートメントを定義する機能を追加します。これに相当するステートメントは JPQL にはありません。HQL の **INSERT** ステートメントの BNF は次の通りです。

```
insert_statement ::= insert_clause select_statement

insert_clause ::= INSERT INTO entity_name (attribute_list)

attribute_list ::= state_field[, state_field ]*
```

attribute_list は、SQL **INSERT** ステートメントの **column specification** と似ています。マップされた継承に関係するエンティティーでは、名前付きエンティティー上で直接定義された属性のみを **attribute_list** で使用することが可能です。スーパークラスプロパティーは許可されず、サブクラスプロパティーは意味がありません。よって、 **INSERT** ステートメントは本質的に非多形となります。

警告

select_statement はあらゆる有効な HQL select クエリになりえますが、戻り型は挿入が想定する型と一致しなければなりません。現在、型の一致はクエリのコンパイル中にチェックされ、チェックはデータベースへ委譲されません。そのため、**同じ** Hibernate タイプではなく**相当** する Hibernate タイプの間で問題が生じる可能性があります。例えば、データベースによって区別されず、変換処理を行える可能性があっても、**org.hibernate.type.DateType** としてマップされた属性と、**org.hibernate.type.TimestampType** として定義された属性との不一致が問題となる可能性があります。

insert ステートメントは id 属性に対して 2 つのオプションを提供します。1 つ目は、id 属性を **attribute_list** に明示的に指定するオプションで、この場合、値は対応する select 式から取得されます。2 つ目は **attribute_list** に指定しないオプションで、この場合生成された値が使用されます。2 つ目のオプションは、「データベース内」で操作する id ジェネレーターを使用する場合のみ選択可能です。このオプションを「インメモリ」タイプのジェネレーターで使用すると構文解析中に例外が生じます。

insert ステートメントは楽観的ロックの属性に対しても 2 つのオプションを提供します。1 つ目は **attribute_list** に属性を指定するオプションで、この場合、値は対応する select 式から取得されます。2 つ目は **attribute_list** に指定しないオプションで、この場合、対応する **org.hibernate.type.VersionType** によって定義される **seed value** が使用されます。

例11.3 INSERT クエリステートメントの例

```
String hqlInsert = "insert into DelinquentAccount (id, name) select
c.id, c.name from Customer c where ...";
int createdEntities = s.createQuery( hqlInsert ).executeUpdate();
```

バグを報告する

11.4.4. FROM 節について

FROM 節の役割は、他のクエリが使用できるオブジェクトモデルタイプの範囲を定義することです。また、他のクエリが使用できる「ID 変数」もすべて定義します。

バグを報告する

11.4.5. WITH 節について

HQL は **WITH** 節を定義し、結合条件を限定します。これは HQL に固有の機能で、JPQL はこの機能を定義しません。

例11.4 with-clause 結合の例

```
select distinct c
from Customer c
    left join c.orders o
```

```
        with o.value > 5000.00
```

生成された SQL では、**with clause** の条件が生成された SQL の **on clause** の一部となりますが、本項の他のクエリでは HQL/JPQL の条件が生成された SQL の **where clause** の一部となることが重要な違いです。この例に特有の違いは重要ではないでしょう。さらに複雑なクエリでは、**with clause** が必要になることがあります。

明示的な結合は、アソシエーションまたはコンポーネント/埋め込み属性を参照することがあります。コンポーネント/埋め込み属性では、結合は単純に論理的で、物理 (SQL) 結合へ相関がありません。

バグを報告する

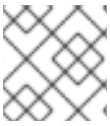
11.4.6. 一括更新、一括送信、および一括削除について

Hibernate では、Data Manipulation Language (DML) を使用して、マップ済みデータベースのデータを直接、一括挿入、一括更新、および一括削除できます (Hibernate Query Language を使用)。

警告

DML を使用すると、オブジェクト/リレーショナルマッピングに違反し、オブジェクトの状態に影響が出ることがあります。オブジェクトの状態はメモリーでは変わりません。DML を使用することにより、基礎となるデータベースで実行された操作に応じて、メモリー内オブジェクトの状態は影響を受けません。DML を使用する場合、メモリー内データは注意を払って使用する必要があります。

UPDATE ステートメントと DELETE ステートメントの擬似構文は **(UPDATE | DELETE) FROM? EntityName (WHERE where_conditions)?** です。



注記

FROM キーワードと **WHERE Clause** はオプションです。

UPDATE ステートメントまたは DELETE ステートメントの実行結果は、実際に影響 (更新または削除) を受けた行の数です。

例11.5 一括更新ステートメントの例

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Company set name = :newName where name =
:oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
```

```
        .executeUpdate();
tx.commit();
session.close();
```

例11.6 一括削除ステートメントの例

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Company where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

Query.executeUpdate() メソッドにより返された **int** 値は、操作で影響を受けたデータベース内のエンティティー数を示します。

内部的に、データベースは複数の SQL ステートメントを使用して DML 更新または削除の要求に対する操作を実行することがあります。多くの場合、これは、更新または削除する必要があるテーブルと結合テーブル間に存在する関係のためです。

たとえば、上記の例のように削除ステートメントを発行すると、**oldName** で指定された会社用の **Company** テーブルだけでなく、結合テーブルに対しても削除が実行されることがあります。したがって、Employee テーブルとの関係が BiDirectional ManyToMany である Company テーブルで、以前の例の正常な実行結果として、対応する結合テーブル **Company_Employee** から複数の行が失われます。

上記の **int deletedEntries** 値には、この操作により影響を受けたすべての行 (結合テーブルの行を含む) の数が含まれます。

INSERT ステートメントの擬似構文は **INSERT INTO EntityName properties_list select_statement** です。

注記

INSERT INTO ... SELECT ... form のみサポートされ、INSERT INTO ... VALUES ... form はサポートされません。

例11.7 一括挿入ステートメントの例

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into Account (id, name) select c.id, c.name
from Customer c where ...";
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

- SELECT ステートメントを介して **id** 属性の値を提供しない場合は、基礎となるデータベースが自動生成されたキーをサポートする限り、ユーザーに対して ID が生成されます。この一括挿入操作の戻り値は、データベースで実際に作成されたエントリーの数です。

バグを報告する

11.4.7. コレクションメンバーの参照について

コレクション値 (collection-valued) アソシエーションへの参照は、実際はコレクションの**値**を参照します。

例11.8 コレクション参照の例

```
select c
from Customer c
    join c.orders o
    join o.lineItems l
    join l.product p
where o.status = 'pending'
  and p.status = 'backorder'

// alternate syntax
select c
from Customer c,
    in(c.orders) o,
    in(o.lineItems) l
    join l.product p
where o.status = 'pending'
  and p.status = 'backorder'
```

この例では、**Customer#orders** アソシエーションの要素タイプであるオブジェクトモデルタイプ **Order** を ID 変数 **o** が実際に参照します。

更にこの例には、**IN** 構文を使用してコレクションアソシエーション結合を指定する代替の構文があります。構文は両方同等です。アプリケーションが使用する構文は任意に選択できます。

バグを報告する

11.4.8. 限定パス式について

コレクション値 (collection-valued) のアソシエーションは、実際にはそのコレクションの**値**を参照すると前項で説明しました。コレクションのタイプを基に、明示的な限定式のセットも使用可能です。

表11.9 限定パス式

式	説明
VALUE	コレクション値を参照します。限定子を指定しないことと同じです。目的を明示的に表す場合に便利です。コレクション値 (collection-valued) の参照のすべてのタイプに対して有効です。

式	説明
INDEX	HQL ルールによると、マップキーまたはリストの場所 (OrderColumn の値) へ参照するよう **javax.persistence.OrderColumn** を指定するマップとリストに対して有効です。JPQL では List の使用に対して確保され、MAP に対して **KEY** を追加します。JPA プロバイダーの移植性に関心があるアプリケーションは、この違いに注意する必要があります。
KEY	マップに対してのみ有効です。マップのキーを参照します。キー自体がエンティティーである場合、更にナビゲートすることが可能です。
ENTRY	マップに対してのみ有効です。マップの論理 **java.util.Map.Entry** タプル (キーと値の組み合わせ) を参照します。**ENTRY** は終端パスとしてのみ有効で、select 節のみで有効になります。

例11.9 限定コレクション参照の例

```
// Product.images is a Map<String,String> : key = a name, value = file
path

// select all the image file paths (the map value) for Product#123
select i
from Product p
    join p.images i
where p.id = 123

// same as above
select value(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names (the map key) for Product#123
select key(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names and file paths (the 'Map.Entry') for
Product#123
select entry(i)
from Product p
    join p.images i
where p.id = 123

// total the value of the initial line items for all orders for a
customer
select sum( li.amount )
from Customer c
```

```
        join c.orders o
        join o.lineItems li
where c.id = 123
  and index(li) = 1
```

バグを報告する

11.4.9. スカラー関数について

HQL は、使用される基盤のデータに関係なく使用できる標準的な関数の一部を定義します。また、HQL は方言やアプリケーションによって定義された追加の関数も理解することができます。

バグを報告する

11.4.10. HQL の標準化された関数

使用される基盤のデータベースに関係なく HQL で使用できる関数は次の通りです。

表11.10 HQL の標準化された関数

関数	説明
BIT_LENGTH	バイナリデータの長さを返します。
CAST	SQL キャストを実行します。キャストターゲットが使用する Hibernate マッピングタイプの名前を付けるはずです。詳細はデータタイプに関する章を参照してください。
EXTRACT	datetime 値で SQL の抽出を実行します。抽出により、datetime 値の一部が抽出されます (年など)。以下の省略形を参照してください。
SECOND	秒を抽出する抽出の省略形。
MINUTE	分を抽出する抽出の省略形。
HOUR	時間を抽出する抽出の省略形。
DAY	日を抽出する抽出の省略形。
MONTH	月を抽出する抽出の省略形。
YEAR	年を抽出する抽出の省略形。
STR	値を文字データとしてキャストする省略形。

アプリケーション開発者は独自の関数セットを提供することもできます。通常、カスタム SQL 関数か SQL スニペットのエイリアスで表します。このような関数は、**org.hibernate.cfg.Configuration** の **addSqlFunction** メソッドを使用して宣言します。

バグを報告する

11.4.11. 連結演算について

HQL は、連結 (**CONCAT**) 関数をサポートするだけでなく、連結演算子も定義します。連結演算子は JPQL によっては定義されないため、移植可能なアプリケーションでは使用しないでください。連結演算子は SQL の連結演算子である **||** を使用します。

例11.10 連結演算の例

```
select 'Mr. ' || c.name.first || ' ' || c.name.last
from Customer c
where c.gender = Gender.MALE
```

バグを報告する

11.4.12. 動的インスタンス化について

select 節でのみ有効な特別な式タイプがありますが、Hibernate では「動的インスタンス化」と呼びます。JPQL はこの機能の一部をサポートし、「コンストラクター式」と呼びます。

例11.11 動的インスタンス化の例 - コンストラクター

```
select new Family( mother, mate, offspr )
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

Object[] に対処せずに、クエリの結果として返されるタイプセーフの Java オブジェクトで値をラッピングします。クラス参照は完全修飾する必要があり、一致するコンストラクターがなければなりません。

ここでは、クラスをマッピングする必要はありません。エンティティーを表す場合、結果となるインスタンスは NEW ステートで返されます (管理されません)。

この部分は JPQL もサポートします。HQL は他の「動的インスタンス化」もサポートします。始めに、スカラーの結果に対して Object[] ではなくリストを返すよう、クエリで指定できます。

例11.12 動的インスタンス化の例 - リスト

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

このクエリの結果は、 List<Object[]> ではなく List<List> になります。

また、HQL はマップにおけるスカラーの結果のラッピングもサポートします。

例11.13 動的インスタンス化の例 - マップ

```
select new map( mother as mother, offspr as offspr, mate as mate )
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr

select new map( max(c.bodyWeight) as max, min(c.bodyWeight) as min,
count(*) as n )
from Cat cxt"/>
```

このクエリの結果は List<Object[]> ではなく List<Map<String,Object>> になります。マップのキーは select 式へ提供されたエイリアスによって定義されます。

バグを報告する

11.4.13. HQL 述語について

述語は where 節、having 節、および検索 case 式の基盤を形成します。これらは式で、通常は **TRUE** または **FALSE** の真理値で解決しますが、NULL が関係するブール値の比較は**UNKNOWN** で解決します。

HQL 述語

NULL 述語

NULL の値をチェックします。基本的な属性参照、エンティティー参照、およびパラメーターへ適用できます。HQL はコンポーネント/埋め込み可能タイプへの適用も許可します。

例11.14 NULL チェックの例

```
// select everyone with an associated address
select p
from Person p
where p.address is not null

// select everyone without an associated address
select p
from Person p
where p.address is null
```

LIKE 述語

文字列値で LIKE 比較を実行します。構文は次の通りです。

```
like_expression ::=
   string_expression
   [NOT] LIKE pattern_value
   [ESCAPE escape_character]
```

セマンティックは SQL の LIKE 式に従います。**pattern_value** は、**string_expression** で一致

を試みるパターンです。SQL と同様に、**pattern_value** に「_」や「%」をワイルドカードとして使用できます。意味も同じで、「_」はあらゆる 1 つの文字と一致し、「%」はあらゆる数の文字と一致します。

任意の **escape_character** は、**pattern_value** の「_」や「%」をエスケープするために使用するエスケープ文字を指定するために使用されます。「_」や「%」が含まれるパターンを検索する必要がある場合に役立ちます。

例11.15 LIKE 述語の例

```
select p
from Person p
where p.name like '%Schmidt'

select p
from Person p
where p.name not like 'Jingleheimmer%'

// find any with name starting with "sp_"
select sp
from StoredProcedureMetadata sp
where sp.name like 'sp|_%' escape '|'
```

BETWEEN 述語

SQL の **BETWEEN** 式と同様です。値が他の 2 つの値の間にあることを評価するため実行します。演算対象はすべて比較可能な型を持つ必要があります。

例11.16 BETWEEN 述語の例

```
select p
from Customer c
    join c.paymentHistory p
where c.id = 123
  and index(p) between 0 and 9

select c
from Customer c
where c.president.dateOfBirth
        between {d '1945-01-01'}
            and {d '1965-01-01'}

select o
from Order o
where o.total between 500 and 5000

select p
from Person p
where p.name between 'A' and 'E'
```

バグを報告する

11.4.14. 関係比較について

比較には比較演算子 (=, >, >=, <, <=, <>]>) の1 つが関与します。また、HQL は <![CDATA[<> の比較演算子の同義として != を定義します。演算対象は同じ型でなければなりません。

例11.17 関係比較の例

```
// numeric comparison
select c
from Customer c
where c.chiefExecutive.age < 30

// string comparison
select c
from Customer c
where c.name = 'Acme'

// datetime comparison
select c
from Customer c
where c.inceptionDate < {d '2000-01-01'}

// enum comparison
select c
from Customer c
where c.chiefExecutive.gender = com.acme.Gender.MALE

// boolean comparison
select c
from Customer c
where c.sendEmail = true

// entity type comparison
select p
from Payment p
where type(p) = WireTransferPayment

// entity value comparison
select c
from Customer c
where c.chiefExecutive = c.chiefTechnologist
```

比較には、サブクエリ限定子である **ALL**、 **ANY**、 **SOME** も関与します。**SOME** と **ANY** は同義です。

サブクエリの結果にあるすべての値に対して比較が **true** である場合、**ALL** 限定子は **true** に解決されます。サブクエリの結果が空の場合は **false** に解決されます。

例11.18 ALL サブクエリ比較限定子の例

```
// select all players that scored at least 3 points
// in every game.
select p
from Player p
```

```
where 3 > all (
   select spg.points
   from StatsPerGame spg
   where spg.player = p
)
```

サブクエリの結果にある値の一部 (最低でも 1 つ) に対して比較が true の場合、**ANY** または **SOME** 限定子は true に解決されます。サブクエリの結果が空である場合、false に解決されます。

バグを報告する

11.4.15. IN 述語

IN 述語は、値のリストに特定の値があることを確認するチェックを行います。構文は次の通りです。

```
in_expression ::= single_valued_expression
            [NOT] IN single_valued_list

single_valued_list ::= constructor_expression |
            (subquery) |
            collection_valued_input_parameter

constructor_expression ::= (expression[, expression]*)
```

single_valued_expression のタイプと **single_valued_list** の各値は一致しなければなりません。JPQL は有効なタイプを文字列、数字、日付、時間、タイムスタンプ、列挙型に限定します。JPQL では、 **single_valued_expression** は下記のみを参照できます。

- 簡単な属性を表す「ステートフィールド」。アソシエーションとコンポーネント/埋め込み属性を明確に除外します。
- エンティティータイプの式。

HQL では、**single_valued_expression** はさらに広範囲の式タイプを参照することが可能です。単一値のアソシエーションは許可されます。コンポーネント/埋め込み属性も許可されますが、この機能は、基礎となるデータベースのタプルまたは「行値コンストラクター構文」へのサポートのレベルに依存します。また、HQL は値タイプを制限しませんが、基礎となるデータベースのベンダーによってはサポートが制限されるタイプがあることをアプリケーション開発者は認識しておいたほうがよいでしょう。これが JPQL の制限の主な原因となります。

値のリストは複数の異なるソースより取得することが可能です。**constructor_expression** と **collection_valued_input_parameter** では、空の値のリストは許可されず、最低でも 1 つの値が含まれなければなりません。

例11.19 IN 述語の例

```
select p
from Payment p
where type(p) in (CreditCardPayment, WireTransferPayment)

select c
from Customer c
where c.hqAddress.state in ('TX', 'OK', 'LA', 'NM')
```

```
select c
from Customer c
where c.hqAddress.state in ?

select c
from Customer c
where c.hqAddress.state in (
    select dm.state
    from DeliveryMetadata dm
    where dm.salesTax is not null
)

// Not JPQL compliant!
select c
from Customer c
where c.name in (
    ('John','Doe'),
    ('Jane','Doe')
)

// Not JPQL compliant!
select c
from Customer c
where c.chiefExecutive in (
    select p
    from Person p
    where ...
)
```

バグを報告する

11.4.16. HQL の順序付けについて

クエリの結果を順序付けすることも可能です。**ORDER BY** 節を使用して、結果を順序付けするために使用される選択値を指定します。**order-by** 節の一部として有効な式タイプには以下が含まれます。

- ステートフィールド
- コンポーネント/埋め込み可能属性
- 算術演算や関数などのスカラー式。
- 前述の式タイプのいずれかに対する **select** 節に宣言された ID 変数。

HQL は、**order-by** 節で参照されたすべての値が **select** 節で名付けされることを強制しませんが、JPQL では必要となります。データベースの移植性を要求するアプリケーションは、**select** 節で参照されない **order-by** 節の参照値をサポートしないデータベースがあることを認識する必要があります。

order-by の各式は、**ASC** (昇順) または **DESC** (降順) で希望の順序を示し限定することができます。

例11.20 ORDER BY の例

```
// legal because p.name is implicitly part of p
```

```
select p
from Person p
order by p.name

select c.id, sum( o.total ) as t
from Order o
    inner join o.customer c
group by c.id
order by t
```

バグを報告する

11.5. HIBERNATE サービス

11.5.1. Hibernate サービスについて

サービスは、さまざまな機能タイプのプラグ可能な実装を Hibernate に提供するクラスです。サービスは特定のサービスコントラクトインターフェースの実装です。インターフェースはサービスロールとして知られ、実装クラスはサービス実装として知られています。通常、ユーザーはすべての標準的なサービスロールの代替実装へプラグインできます (オーバーライド)。また、サービスロールのベースセットを越えた追加サービスを定義できます (拡張)。

バグを報告する

11.5.2. サービスコントラクトについて

マーカーインターフェース **`org.hibernate.service.Service`** を実装することがサービスの基本的な要件になります。Hibernate は基本的なタイプセーフのために内部でこのインターフェースを使用します。

起動と停止の通知を受け取るため、サービスは **`org.hibernate.service.spi.Startable`** および **`org.hibernate.service.spi.Stoppable`** インターフェースを任意で実装することもできます。その他に、JMX 統合が有効になっている場合に JMX でサービスを管理可能としてマーク付けする **`org.hibernate.service.spi.Manageable`** という任意のサービスコントラクトがあります。

バグを報告する

11.5.3. サービス依存関係のタイプ

サービスは、以下の 2 つの方法のいずれかを使用して、他のサービスに依存関係を宣言することができます。

`@org.hibernate.service.spi.InjectService`

単一のパラメーターを許可するサービス実装上のすべてのメソッドと、**`@InjectService`** アノテーションが付けられているメソッドは、他のサービスの挿入を要求していると見なされます。

デフォルトではメソッドパラメーターのタイプは、挿入されるサービスロールであると想定されます。パラメータータイプがサービスロールではない場合、**`InjectService`** の **`serviceRole`** 属性を使用してロールを明示的に指定する必要があります。

デフォルトでは、挿入されたサービスは必須のサービスであると見なされます。そのため、名前付けされた依存サービスがない場合、起動に失敗します。挿入されるサービスが任意のサービスであ

る場合、**InjectService** の **required** 属性を **false** として宣言する必要があります (デフォルトは **true** です)。

org.hibernate.service.spi.ServiceRegistryAwareService

2 つ目の方法は、単一の **injectServices** メソッドを宣言する任意のサービスインターフェース **org.hibernate.service.spi.ServiceRegistryAwareService** をサービスが実装する方法です。

起動中、Hibernate は **org.hibernate.service.ServiceRegistry** 自体をこのインターフェースが実装するサービスに挿入します。その後、サービスは **ServiceRegistry** 参照を使用して、必要な他のサービスを見つけることができます。

バグを報告する

11.5.4. ServiceRegistry

11.5.4.1. ServiceRegistry について

サービス自体を除いた中央サービス API は **org.hibernate.service.ServiceRegistry** インターフェースです。サービスレジストリーの主な目的は、サービスを保持管理し、サービスへのアクセスを提供することです。

サービスレジストリーは階層的です。レジストリーのサービスは、同じレジストリーおよび親レジストリーにあるサービスへ依存したり利用したりすることが可能です。

org.hibernate.service.ServiceRegistryBuilder を使用して **org.hibernate.service.ServiceRegistry** インスタンスをビルドします。

例11.21 ServiceRegistryBuilder を使用した ServiceRegistry の作成

```
ServiceRegistryBuilder registryBuilder = new ServiceRegistryBuilder(
bootstrapServiceRegistry );
    ServiceRegistry serviceRegistry =
registryBuilder.buildServiceRegistry();
```

バグを報告する

11.5.5. カスタムサービス

11.5.5.1. カスタムサービスについて

ビルドされた **org.hibernate.service.ServiceRegistry** は不変であると見なされます。サービス自体は再設定を許可することもありますが、ここで言う不変とはサービスの追加や置換を意味します。そのため **org.hibernate.service.ServiceRegistryBuilder** によって提供される別のロールは、生成された **org.hibernate.service.ServiceRegistry** に格納されるサービスを微調整できるようにします。

カスタムサービスについて **org.hibernate.service.ServiceRegistryBuilder** に通知する方法は 2 つあります。

- **org.hibernate.service.spi.BasicServiceInitiator** クラスを実装してサービスクラスの要求に応じた構築を制御し、**addInitiator** メソッドより **org.hibernate.service.ServiceRegistryBuilder** へ追加します。
- サービスクラスをインスタンス化し、**addService** メソッドより **org.hibernate.service.ServiceRegistryBuilder** へ追加します。

サービスを追加する方法とイニシエーターを追加する方法はいずれも、レジストリーの拡張 (新しいサービスロールの追加) やサービスのオーバーライド (サービス実装の置換) に対して有効です。

例11.22 ServiceRegistryBuilder を用いた既存サービスのカスタマーサービスへの置き換え

```
   ServiceRegistryBuilder registryBuilder = new ServiceRegistryBuilder(
bootstrapServiceRegistry );
   serviceRegistryBuilder.addService( JdbcServices.class, new
FakeJdbcService() );
   ServiceRegistry serviceRegistry =
registryBuilder.buildServiceRegistry();


   public class FakeJdbcService implements JdbcServices{

       @Override
       public ConnectionProvider getConnectionProvider() {
           return null;
       }

       @Override
       public Dialect getDialect() {
           return null;
       }

       @Override
       public SqlStatementLogger getSqlStatementLogger() {
           return null;
       }

       @Override
       public SqlExceptionHelper getSqlExceptionHelper() {
           return null;
       }

       @Override
       public ExtractedDatabaseMetaData getExtractedMetaDataSupport() {
           return null;
       }

       @Override
       public LobCreator getLobCreator(LobCreationContext
lobCreationContext) {
           return null;
       }

       @Override
       public ResultSetWrapper getResultSetWrapper() {
           return null;
```

```
        }

        @Override
        public JdbcEnvironment getJdbcEnvironment() {
            return null;
        }
    }
```

バグを報告する

11.5.6. ブートストラップレジストリ

11.5.6.1. ブートストラップレジストリーについて

ブートストラップレジストリーは、動作するために絶対に使用可能でなければならないサービスを保持します。主なサービスは **ClassLoaderService** で、代表的な例になります。設定ファイルの解決にもクラスローディングサービス (リソースのルックアップ) へのアクセスが必要になります。通常の使用ではこれがルートレジストリーになります。

ブートストラップレジストリーのインスタンスは **org.hibernate.service.BootstrapServiceRegistryBuilder** クラスを使用して構築されます。

バグを報告する

11.5.6.2. BootstrapServiceRegistryBuilder の使用

例11.23 BootstrapServiceRegistryBuilder の使用

```
BootstrapServiceRegistry bootstrapServiceRegistry = new
BootstrapServiceRegistryBuilder()
        // pass in org.hibernate.integrator.spi.Integrator instances
which are not
        // auto-discovered (for whatever reason) but which should be
included
        .with( anExplicitIntegrator )
        // pass in a class-loader Hibernate should use to load
application classes
        .withApplicationClassLoader(
anExplicitClassLoaderForApplicationClasses )
        // pass in a class-loader Hibernate should use to load
resources
        .withResourceClassLoader( anExplicitClassLoaderForResources )
        // see BootstrapServiceRegistryBuilder for rest of available
methods
        ...
        // finally, build the bootstrap registry with all the above
options
        .build();
```

バグを報告する

11.5.6.3. BootstrapRegistry サービス

`org.hibernate.service.classloading.spi.ClassLoaderService`

Hibernate は ClassLoaders と対話する必要がありますが、Hibernate (またはライブラリ) が ClassLoaders と対話する方法は、アプリケーションをホストするランタイム環境によって異なります。アプリケーションサーバー、OSGi コンテナ、およびその他のモジュラークラスローディングシステムによって、クラスローディングの要件が非常に具体的になります。このサービスは、この複雑な環境より抽象化を Hibernate に提供しますが、単一のスワップ可能なコンポーネントを用いることも重要な点になります。

ClassLoader との対話では、Hibernate に以下の機能が必要になります。

- アプリケーションクラスを見つける機能
- 統合クラスを見つける機能
- リソース (プロパティーファイル、xml ファイルなど) を見つける機能
- **`java.util.ServiceLoader`** をロードする機能

注記

現在、アプリケーションクラスをロードする機能と統合クラスをロードする機能は、サービス上の 1 つの「ロードクラス」機能として組み合わされていますが、今後のリリースで変更になる可能性があります。

`org.hibernate.integrator.spi.IntegratorService`

Hibernate との統合に必要なアプリケーションやアドオンなどすべてのもの。各インテグレーターの代わりに、必要とする各統合部分の登録をコーディネートするよう、アプリケーションなどに要求するために使用されます。

このサービスはディスカバリの側面に注目します。**`org.hibernate.service.classloading.spi.ClassLoaderService`** によって提供される標準の Java **`java.util.ServiceLoader`** 機能を使用して、**`org.hibernate.integrator.spi.Integrator`** コントラクトの実装をディスカバリします。

インテグレーターは **`/META-INF/services/org.hibernate.integrator.spi.Integrator`** という名前のファイルを定義し、クラスパス上で使用できるようにします。**`java.util.ServiceLoader`** は詳細にこのファイルの形式をカバーしますが、本質的に **`org.hibernate.integrator.spi.Integrator`** を行ごとに実装する FQN によるクラスを一覧表示します。

バグを報告する

11.5.7. SessionFactory レジストリ

11.5.7.1. SessionFactory レジストリ

すべてのレジストリタイプのインスタンスを指定の **`org.hibernate.SessionFactory`** のターゲットとして扱うことが最良の方法ですが、このグループのサービスのインスタンスは明示的に 1 つの **`org.hibernate.SessionFactory`** に属します。

起動する必要がある場合、違いはタイミングになります。一般的に起動される **org.hibernate.SessionFactory** にアクセスする必要があります。この特別なレジストリは **org.hibernate.service.spi.SessionFactoryServiceRegistry** です。

バグを報告する

11.5.7.2. SessionFactory サービス

org.hibernate.event.service.spi.EventListenerRegistry

説明

イベントリスナーを管理するサービス。

イニシエーター

org.hibernate.event.service.internal.EventListenerServiceInitiator

実装

org.hibernate.event.service.internal.EventListenerRegistryImpl

バグを報告する

11.5.8. インテグレーター

11.5.8.1. インテグレーター

org.hibernate.integrator.spi.Integrator の目的は、機能する SessionFactory のビルドプロセスを開発者がフックできるようにする簡単な手段を提供することです。**org.hibernate.integrator.spi.Integrator** インターフェースは、ビルドプロセスをフックできるようにする **integrate** と、終了する SessionFactory をフックできるようにする **disintegrate** の 2 つのメソッドを定義します。

注記

org.hibernate.cfg.Configuration の代わりに **org.hibernate.metamodel.source.MetadataImplementor** を許可するオーバーロードした形式の **integrate** は、**org.hibernate.integrator.spi.Integrator** で定義される 3 つ目のメソッドになります。

IntegratorService によって提供されるディスカバリ以外に、BootstrapServiceRegistry のビルド時にアプリケーションはインテグレーターを手動で登録することができます。

バグを報告する

11.5.8.2. インテグレーターのユースケース

現在、**org.hibernate.integrator.spi.Integrator** の主なユースケースは、イベントリスナーの登録とサービスの提供になります (**org.hibernate.integrator.spi.ServiceContributingIntegrator** を参照)。5.0 では、オブジェクトとリレーショナルモデルとの間のマッピングを記述するメタモデルを変更できるようにするための拡張を計画しています。

例11.24 イベントリスナーの登録

```
public class MyIntegrator implements
org.hibernate.integrator.spi.Integrator {

    public void integrate(
            Configuration configuration,
            SessionFactoryImplementor sessionFactory,
            SessionFactoryServiceRegistry serviceRegistry) {
        // As you might expect, an EventListenerRegistry is the thing
with which event listeners are registered  It is a
        // service so we look it up using the service registry
        final EventListenerRegistry eventListenerRegistry =
serviceRegistry.getService( EventListenerRegistry.class );

        // If you wish to have custom determination and handling of
"duplicate" listeners, you would have to add an
        // implementation of the
org.hibernate.event.service.spi.DuplicationStrategy contract like this
        eventListenerRegistry.addDuplicationStrategy(
myDuplicationStrategy );

        // EventListenerRegistry defines 3 ways to register listeners:
        //     1) This form overrides any existing registrations with
        eventListenerRegistry.setListeners( EventType.AUTO_FLUSH,
myCompleteSetOfListeners );
        //     2) This form adds the specified listener(s) to the
beginning of the listener chain
        eventListenerRegistry.prependListeners( EventType.AUTO_FLUSH,
myListenersToBeCalledFirst );
  //     3) This form adds the specified listener(s) to the end of the
listener chain
        eventListenerRegistry.appendListeners( EventType.AUTO_FLUSH,
myListenersToBeCalledLast );
    }
}
```

バグを報告する

11.6. BEAN VALIDATION

11.6.1. Bean 検証について

Bean 検証あるいは JavaBeans 検証は、Java オブジェクトのデータを検証するモデルです。このモデルは、同梱でカスタムのアノテーション制約を使い、アプリケーションデータの整合性を保ちます。この仕様は http://jcp.org/en/jsr/detail?id=303 に文書でまとめられています。

Hibernate バリデーターは Bean 検証の JBoss Enterprise Application Platform 実装で、JSR の参照実装でもあります。

JBoss Enterprise Application Platform 6 は JSR303 の Bean 検証に完全準拠しています。Hibernate バリデーターはこの仕様に対しさらに機能を提供しています。

Bean 検証を利用するには、**bean-validation** クイックスタートの例 「Java EE クイックスタートサンプルへのアクセス」 を参照してください。

バグを報告する

11.6.2. Hibernate バリデーター

Hibernate バリデーターは、JSR 303 - Bean Validation の参照実装です。

Bean 検証は、Java オブジェクトデータを検証するモデルをユーザーに提供します。詳細については、「Bean 検証について」と 「バリデーション制約について」を参照してください。

バグを報告する

11.6.3. バリデーション制約

11.6.3.1. バリデーション制約について

バリデーション制約とは、フィールド、プロパティ、あるいは Bean といったJava 要素に適用するルールのことです。制約は通常、制限を設定する際に利用する属性の組み合わせのことです。定義済みの制約がありますが、カスタムの制約も作成可能です。各制約は、アノテーション形式で表現していきます。

Hibernate Validator 用の同梱のバリデーション制約は、「Hibernate Validator の制約」に一覧表示されています。

詳細は 「Hibernate バリデーター」および 「Bean 検証について」 を参照してください。

バグを報告する

11.6.3.2. JBoss Developer Studio で制約アノテーションを作成

概要

このタスクでは、Java アプリケーションで利用できるように、JBoss Developer Studio で制約アノテーションを作成するプロセスを説明します。

要件

- 「JBoss Developer Studio の起動」

手順11.5 制約アノテーションを作成する

1. JBoss Developer Studio で Java プロジェクトを開きます。
2. **データセットの作成**
   制約アノテーションには、許容値を定義するデータセットが必要です。
   a. **[Project Explorer]** パネルのプロジェクトの root フォルダーを右クリックします。
   b. **[New]** → **[Enum]** を選択します。
   c. 以下の要素を設定してください。
      - **Package:**

- **Name:**

d. **[Add...]** ボタンをクリックし必要なインターフェースを追加します。

e. **[Finish]** をクリックし、ファイルを作成します。

f. データセットに値を追加し **[Save]** をクリックします。

例11.25 データセットの例

```
package com.example;

public enum CaseMode {
    UPPER,
    LOWER;
}
```

3. **アノテーションファイルの作成**
新しい Java クラスを作成します。詳細については、「JBoss Developer Studio での新しい Java クラスの作成」を参照してください。

4. 制約アノテーションを設定し **[Save]** をクリックします。

例11.26 制約アノテーションファイルの例

```
package com.mycompany;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = CheckCaseValidator.class)
@Documented
public @interface CheckCase {

    String message() default "
{com.mycompany.constraints.checkcase}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    CaseMode value();

}
```

結果

許容値のあるカスタムの制約アノテーションが作成され、Java プロジェクトで使用することができます。

バグを報告する

11.6.3.3. JBoss Developer Studio での新しい Java クラスの作成

要件

- 「JBoss Developer Studio の起動」

概要

本トピックでは、JBoss Developer Studio を使用して既存の Java プロジェクトに対して Java クラスを作成する手順について説明します。

手順11.6 新しい Java クラスの作成

1. **[Project Explorer]** パネルのプロジェクトの root フォルダーを右クリックします。
2. **[New]** → **[Class]** と選択します。
3. 以下の要素を設定してください。
   - **Package:**
   - **Name:**
4. **任意: インターフェースの追加**
   a. **[Add...]** をクリックします。
   b. インターフェース名の検索
   c. 正しいインターフェースの選択
   d. 必要なインターフェースごとに手順 2 と 3 を繰り返す
   e. **[Add]** をクリックします。
5. **[Finish]** をクリックし、ファイルを作成します。

結果

設定の準備が整った新しい Java クラスがプロジェクト内に作成されます。

バグを報告する

11.6.3.4. Hibernate Validator の制約

表11.11 同梱の制約

アノテーション	適用先	ランタイムチェック	Hibernate Metadata の影響
@Length(min=, max=)	プロパティ (文字列)	文字列の長さが指定の範囲とマッチしているか確認	カラムの長さを最大に設定
@Max(value=)	プロパティ (数字あるいは数字の文字列表現)	値が最大値以下であるか確認	カラムに check 制約を追加
@Min(value=)	プロパティ (数字あるいは数字の文字列表現)	値が最小値以上であるか確認	カラムに check 制約を追加
@NotNull	プロパティ	値が null でないか確認	カラムが null でないか確認
@NotEmpty	プロパティ	文字列が null あるいは空でないか確認。接続が null あるいは空でないか確認	カラムが (文字列に対し) null でないか確認
@Past	プロパティ (日付あるいはカレンダー)	日付が過去のものかを確認	カラムに check 制約を追加
@Future	プロパティ (日付あるいはカレンダー)	日付が未来のものかを確認	なし
@Pattern(regex="regexp", flag=) or @Patterns({@Pattern(...)})	プロパティ (文字列)	プロパティが一致フラグを渡す正規表現に一致しているか確認 (`java.util.regex.Pattern` 参照)	なし
@Range(min=, max=)	プロパティ (数字あるいは数字の文字列表現)	最小値以上、最大値以下であるか確認	カラムに check 制約を追加
@Size(min=, max=)	プロパティ (アレイ、コレクション、マップ)	要素サイズが最小値以上、最大値以下であるかを確認	なし
@AssertFalse	プロパティ	メソッドが false と評価しているよう確認 (アノテーションでなくコードで制約が表現されている場合に便利)	なし
@AssertTrue	プロパティ	メソッドが true と評価しているよう確認 (アノテーションでなくコードで制約が表現されている場合に便利)	なし

アノテーション	適用先	ランタイムチェック	Hibernate Metadata の影響
@Valid	プロパティ (オブジェクト)	紐付けされたオブジェクトに再帰的にバリデーションを行う。オブジェクトがコレクションかアレイの場合は、要素は再帰的に検証されます。また、オブジェクトがマップの場合、値要素が再帰的に検証されます。	なし
@Email	プロパティ (文字列)	文字列が電子メールアドレスの仕様に準拠するかどうかを確認します。	なし
@CreditCardNumber	プロパティ (文字列)	文字列がクレジットカード番号用に適切にフォーマットされているか確認 (Luhn アルゴリズムの派生)	なし
@Digits(integerDigits=1)	プロパティ (数字あるいは数字の文字列表現)	プロパティが最大 **integerDigits** の整数桁と **fractionalDigits** 少数点以下の桁数を持つ数字であるか確認	カラムの制度とスケールを定義
@EAN	プロパティ (文字列)	文字列が正しくフォーマットされた EAN あるいは UPC-A コードであるか確認	なし

バグを報告する

11.6.4. 設定

11.6.4.1. 検証設定ファイルの例

例11.27 validation.xml

```
<validation-config
xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configurati
on">

    <default-provider>
        org.hibernate.validator.HibernateValidator
```

```
    </default-provider>
    <message-interpolator>

org.hibernate.validator.messageinterpolation.ResourceBundleMessageInterp
olator
    </message-interpolator>
    <constraint-validator-factory>
        org.hibernate.validator.engine.ConstraintValidatorFactoryImpl
    </constraint-validator-factory>

    <constraint-mapping>
        /constraints-example.xml
    </constraint-mapping>

    <property name="prop1">value1</property>
    <property name="prop2">value2</property>
</validation-config>
```

バグを報告する

11.7. ENVERS

11.7.1. Hibernate Envers について

Hibernate Envers は監査およびバージョニングシステムで、永続クラスへの変更履歴をトラッキングする方法を JBoss Enterprise Application Platform に提供します。エンティティーに対する変更の履歴を保存する **@Audited** アノテーションが付けられたエンティティーに対して監査テーブルが作成されます。その後、データの読み出しやクエリが可能になります。

Envers により開発者は次の作業を行うことが可能になります。

- JPA 仕様によって定義されるすべてのマッピングの監査
- JPA 仕様を拡張するすべての Hibernate マッピングの監査
- ネイティブ Hibernate API によりマッピングされた監査エンティティー
- リビジョンエンティティーを用いて各リビジョンのデータをログに記録
- 履歴データのクエリ

バグを報告する

11.7.2. 永続クラスの監査について

JBoss Enterprise Application Platform では Hibernate Envers と **@Audited** アノテーションを使用して永続クラスの監査を行います。アノテーションがクラスに適用されると、エンティティーのリビジョン履歴が保存されるテーブルが作成されます。

クラスに変更が加えられるたびに監査テーブルにエントリが追加されます。エントリーにはクラスへの変更が含まれ、リビジョン番号が付けられます。そのため、変更をロールバックしたり、以前のリビジョンを表示したりすることが可能です。

バグを報告する

11.7.3. 監査ストラテジー

11.7.3.1. 監査ストラテジーについて

監査ストラテジーは、監査情報の永続化、クエリ、格納の方法を定義します。Hibernate Envers には現在 2 つの監査ストラテジが存在します。

デフォルトの監査ストラテジー

このストラテジーは監査データと開始リビジョンを共に永続化します。監査テーブルで挿入、更新、削除された各行については、開始リビジョンの有効性と合わせて、1 つ以上の行が監査テーブルに挿入されます。

監査テーブルの行は挿入後には更新されません。監査情報のクエリはサブクエリを使い監査テーブルの該当行を選択します (これは時間がかかり、インデックス化が困難です)。

妥当性監査ストラテジー

このストラテジーは監査情報の開始リビジョンと最終リビジョンの両方を格納します。監査テーブルで挿入、更新、削除された各行については、開始リビジョンの有効性とあわせて、1 つ以上の行が監査テーブルに挿入されます。

同時に、以前の監査行 (利用可能な場合) の最終リビジョンフィールドがこのリビジョンに設定されます。監査情報に対するクエリーは、サブクエリーの代わりに**開始と最終リビジョンのいずれかを**使用します。つまり、更新の数が増えるため監査情報の永続化には今までより少し時間がかかりますが、監査情報の取得は非常に早くなります。

インデックスを増やすことで改善することも可能です。

監査の詳細については、「永続クラスの監査について」を参照してください。アプリケーションの監査ストラテジーを設定するには、「監査ストラテジーの設定」を参照してください。

バグを報告する

11.7.3.2. 監査ストラテジーの設定

概要

JBoss Enterprise Application Platform 6 によってサポートされる監査ストラテジーにはデフォルト監査ストラテジーと妥当性監査ストラテジーの 2 つがあります。本タスクでは、アプリケーションに対して監査ストラテジーを定義するために必要な手順について取り上げます。

手順11.7 監査ストラテジーの定義

- アプリケーションの **`persistence.xml`** ファイルの **`org.hibernate.envers.audit_strategy`** プロパティーを設定します。このプロパティーが **`persistence.xml`** ファイルで設定されていない場合は、デフォルトの監査ストラテジーが使用されます。

  例11.28 デフォルトの監査ストラテジーの設定

  ```
  <property name="org.hibernate.envers.audit_strategy"
  value="org.hibernate.envers.strategy.DefaultAuditStrategy"/>
  ```

例11.29 妥当性監査ストラテジーの設定

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.ValidityAuditStrategy"/>
```

バグを報告する

11.7.4. エンティティー監査の開始

11.7.4.1. JPA エンティティーへの監査サポートの追加

JBoss Enterprise Application Platform 6 は、「Hibernate Envers について」を行ってエンティティーの監査を使用し、永続クラスの変更履歴を追跡します。本トピックでは、JPA エンティティーへの監査サポートを追加する方法について取り上げます。

手順11.8 JPA エンティティーへの監査サポートの追加

1. 「Envers パラメーターの設定」に従って、デプロイメントに適した使用可能な監査パラメーターを設定します。
2. 監査対象となる JPA エンティティーを開きます。
3. **`org.hibernate.envers.Audited`** インターフェースをインポートします。
4. 監査対象となる各フィールドまたはプロパティーに **`@Audited`** アノテーションを付けます。または、1 度にクラス全体へアノテーションを付けます。

例11.30 2 つのフィールドの監査

```
import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;

@Entity
public class Person {
    @Id
    @GeneratedValue
    private int id;

    @Audited
    private String name;

    private String surname;

    @ManyToOne
    @Audited
    private Address address;
```

```
    // add getters, setters, constructors, equals and hashCode
here
}
```

例11.31 クラス全体の監査

```
import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;

@Entity
@Audited
public class Person {
    @Id
    @GeneratedValue
    private int id;

    private String name;

    private String surname;

    @ManyToOne
    private Address address;

    // add getters, setters, constructors, equals and hashCode
here
}
```

結果

JPA エンティティーの監査が設定されました。変更履歴を保存するため ***Entity*_AUD** と呼ばれるテーブルが作成されます。

バグを報告する

11.7.5. 設定

11.7.5.1. Envers パラメーターの設定

JBoss Enterprise Application Platform 6 は、Hibernate Envers よりエンティティーの監査を使用し、永続クラスの変更履歴を追跡します。ここでは、使用可能な Envers パラメーターの設定について取り上げます。

手順11.9 Envers パラメーターの設定

1. アプリケーションの **persistence.xml** ファイルを開きます。
2. 必要に応じて Envers プロパティーを追加、削除、設定します。使用可能なプロパティーの一覧は 「Envers の設定プロパティー」 を参照してください。

例11.32 Envers パラメーターの例

```
<persistence-unit name="mypc">
  <description>Persistence Unit.</description>
  <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="create-drop" />
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.cache.use_second_level_cache" value="true"
/>
    <property name="hibernate.cache.use_query_cache" value="true" />
    <property name="hibernate.generate_statistics" value="true" />
    <property name="org.hibernate.envers.versionsTableSuffix" value="_V"
/>
    <property name="org.hibernate.envers.revisionFieldName"
value="ver_rev" />
  </properties>
</persistence-unit>
```

結果

アプリケーションのすべての JPA エンティティーに対して監査の設定が行われます。

バグを報告する

11.7.5.2. ランタイム時に監査を有効または無効にする

概要

このタスクでは、ランタイム時にエンティティーバージョン監査を有効または無効にするために必要な設定手順について取り上げます。

手順11.10 監査を有効/無効にする

1. **`AuditEventListener`** クラスをサブクラス化します。
2. Hibernate イベント上で呼び出される次のメソッドを上書きします。
   - onPostInsert
   - onPostUpdate
   - onPostDelete
   - onPreUpdateCollection
   - onPreRemoveCollection
   - onPostRecreateCollection
3. イベントのリスナーとしてサブクラスを指定します。
4. 変更を監査すべきであるか判断します。

5. 変更を監査する必要がある場合は、呼び出しをスーパークラスへ渡します。

バグを報告する

11.7.5.3. 条件付き監査の設定

概要

Hibernate Envers は多くのイベントリスナーを使用して、さまざまな Hibernate イベントに対して監査データを永続化します。Envers jar がクラスパスにある場合、これらのリスナーは自動的に登録されます。このタスクでは、Envers イベントリスナーの一部を上書きして条件付き監査を実装するために必要な手順について取り上げます。

手順11.11 条件付き監査の実装

1. **`persistence.xml`** ファイルで **`hibernate.listeners.envers.autoRegister`** の Hibernate プロパティーを false に設定します。
2. 上書きする各イベントリスナーをサブクラス化します。条件付き監査の論理をサブクラスに置き、監査の実行が必要な場合はスーパーメソッドを呼び出します。
3. **`org.hibernate.envers.event.EnversIntegrator`** と似ている **`org.hibernate.integrator.spi.Integrator`** のカスタム実装を作成します。デフォルトのクラスではなく、手順の 2 で作成したイベントリスナーサブクラスを使用します。
4. jar に **`META-INF/services/org.hibernate.integrator.spi.Integrator`** ファイルを追加します。このファイルにはインターフェースを実装するクラスの完全修飾名が含まれなければなりません。

結果

条件付き監査が設定され、デフォルトの Envers イベントリスナーがオーバーライドされます。

バグを報告する

11.7.5.4. Envers の設定プロパティー

表11.12 エンティティーデータのバージョニング設定パラメーター

プロパティー名	デフォルト値	説明
org.hibernate.envers.audit_table_prefix		監査エンティティーの名前の前に付けられた文字列。監査情報を保持するエンティティーの名前を作成します。
org.hibernate.envers.audit_table_suffix	_AUD	監査情報を保持するエンティティーの名前を作成する監査エンティティーの名前に追加された文字列。たとえば、**`Person`** のテーブル名を持つエンティティーが監査される場合は、Envers により履歴データを格納する **`Person_AUD`** と呼ばれるテーブルが生成されます。

プロパティー名	デフォルト値	説明
org.hibernate.envers.revision_field_name	REV	改訂番号を保持する監査エンティティーのフィールド名。
org.hibernate.envers.revision_type_field_name	REVTYPE	リビジョンタイプを保持する監査エンティティーのフィールド名。現在のリビジョンタイプは、**add**、**mod**、および**del** です。
org.hibernate.envers.revision_on_collection_change	true	このプロパティーは、所有されていない関係フィールドが変更された場合にリビジョンを生成するかどうかを決定します。これは、一対多関係のコレクションまたは一対一関係の **mappedBy** 属性を使用したフィールドのいずれかです。
org.hibernate.envers.do_not_audit_optimistic_locking_field	true	true の場合、オプティミスティックロッキングに使用したプロパティー (**@Version** のアノテーションがついたもの) は自動的に監査から除外されます。
org.hibernate.envers.store_data_at_delete	false	このプロパティーは、ID のみではなく、他の全プロパティーが null とマークされたエンティティーが削除される場合にエンティティーデータをリビジョンに保存すべきかどうかを定義します。このデータは最終リビジョンに存在するため、これは通常必要ありません。最終リビジョンのデータにアクセスする方が簡単で効率的ですが、この場合、削除前にエンティティーに含まれたデータが 2 回保存されることになります。
org.hibernate.envers.default_schema	null (通常のテーブルと同じ)	監査テーブルに使用されるデフォルトのスキーマ名。**@AuditTable(schema="...")** アノテーションを使用してオーバーライドできます。このスキーマがない場合、スキーマは通常のテーブルのスキーマと同じです。
org.hibernate.envers.default_catalog	null (通常のテーブルと同じ)	監査テーブルに使用するデフォルトのカタログ名。**@AuditTable(catalog="...")** アノテーションを使用してオーバーライドできます。このカタログがない場合、カタログは通常のテーブルのカタログと同じです。

プロパティー名	デフォルト値	説明
org.hibernate.envers.audit_strategy	org.hibernate.envers.strategy.DefaultAuditStrategy	このプロパティーは、監査データを永続化する際に使用する監査ストラテジーを定義します。デフォルトでは、エンティティーが変更されたリビジョンのみが保存されます。あるいは、**`org.hibernate.envers.strategy.ValidityAuditStrategy`** が、開始リビジョンと最終リビジョンの両方を保存します。これらは、監査行が有効である場合に定義されます。
org.hibernate.envers.audit_strategy_validity_end_rev_field_name	REVEND	監査エンティティーのリビジョン番号を保持するカラムの名前。このプロパティーは、妥当性監査ストラテジーが使用されている場合のみ有効です。
org.hibernate.envers.audit_strategy_validity_store_revend_timestamp	false	このプロパティーは、データが最後に有効だった最終リビジョンのタイムスタンプを最終リビジョンとともに格納するかどうかを定義します。これは、テーブルパーティショニングを使用することにより、関係データベースから以前の監査レコードを削除する場合に役に立ちます。パーティショニングには、テーブル内に存在する列が必要です。このプロパティーは、**`ValidityAuditStrategy`** が使用される場合にのみ評価されます。
org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name	REVEND_TSTMP	データが有効であった最終リビジョンのタイムスタンプの列名。**`ValidityAuditStrategy`** が使用され、**`org.hibernate.envers.audit_strategy_validity_store_revend_timestamp`** が true と評価された場合のみ使用されます。

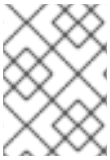
バグを報告する

11.7.6. クエリ

11.7.6.1. 監査情報の読み出し

概要

Hibernate Envers はクエリより監査情報を読み出しする機能を提供します。このトピックではクエリの例を取り上げます。



注記

監査されたデータのクエリは相関サブセレクトが関与するため、多くの場合で **live** データの対応するクエリよりも大幅に処理が遅くなります。

例11.33 特定のリビジョンでクラスのエンティティーをクエリする

このようなクエリのエントリーポイントは次の通りです。

```
AuditQuery query = getAuditReader()
    .createQuery()
    .forEntitiesAtRevision(MyEntity.class, revisionNumber);
```

AuditEntity ファクトリクラスを使用して制約を指定することができます。以下のクエリは、**name** プロパティーが **John** と同等である場合のみエンティティーを選択します。

```
query.add(AuditEntity.property("name").eq("John"));
```

以下のクエリは特定のエンティティーと関連するエンティティーのみを選択します。

```
query.add(AuditEntity.property("address").eq(relatedEntityInstance));
// or
query.add(AuditEntity.relatedId("address").eq(relatedEntityId));
```

結果を順序付けや制限付けしたり、凝集 **(aggregations)** および射影 **(projections)** のセット **(**グループ化を除く**)** を持つことが可能です。以下はフルクエリの例になります。

```
List personsAtAddress = getAuditReader().createQuery()
    .forEntitiesAtRevision(Person.class, 12)
    .addOrder(AuditEntity.property("surname").desc())
    .add(AuditEntity.relatedId("address").eq(addressId))
    .setFirstResult(4)
    .setMaxResults(2)
    .getResultList();
```

例11.34 特定クラスのエンティティーが変更された場合のクエリリビジョン

このようなクエリのエントリーポイントは次の通りです。

```
AuditQuery query = getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true);
```

前の例と同様に、このクエリへ制約を追加することが可能です。このクエリに以下を追加することも可能です。

AuditEntity.revisionNumber()

監査されたエンティティーが修正されたリビジョン番号の制約や射影、順序付けを指定します。

AuditEntity.revisionProperty(propertyName)

監査されたエンティティーが修正されたリビジョンに対応するリビジョンエンティティーのプロパティーの制約や射影、順序付けを指定します。

AuditEntity.revisionType()

リビジョンのタイプ (ADD、MOD、DEL) へのアクセスを提供します。

クエリ結果を必要に応じて調整することが可能です。次のクエリは、リビジョン番号 42 の後に **entityId** ID を持つ **MyEntity** クラスのエンティティーが変更された最小のリビジョン番号を選択します。

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.id().eq(entityId))
    .add(AuditEntity.revisionNumber().gt(42))
    .getSingleResult();
```

リビジョンのクエリはプロパティーを最小化および最大化することも可能です。次のクエリは、特定エンティティーの **actualDate** 値が指定の値よりは大きく、可能な限り小さいリビジョンを選択します。

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    // We are only interested in the first revision
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.property("actualDate").minimize()
        .add(AuditEntity.property("actualDate").ge(givenDate))
        .add(AuditEntity.id().eq(givenEntityId)))
    .getSingleResult();
```

minimize() および **maximize()** メソッドは制約を追加できる基準を返します。最大化または最小化されたプロパティーを持つエンティティーはこの基準を満たさなければなりません。

クエリ作成時に渡されるブール変数パラメーターは 2 つあります。

selectEntitiesOnly

このパラメーターは明示的な射影が設定されていない場合のみ有効です。

true の場合、クエリの結果は指定された制約を満たすリビジョンで変更されたエンティティーの一覧になります。

false の場合、結果は 3 つの要素アレイの一覧になります。最初の要素は変更されたエンティティーインスタンスになります。2 番目の要素はリビジョンデータが含まれるエンティティーになります。カスタムエンティティーが使用されていない場合は **DefaultRevisionEntity** のインスタンスになります。3 つ目の要素アレイはリビジョンのタイプ (ADD、MOD、DEL) になります。

selectDeletedEntities

このパラメーターは、エンティティーが削除されたリビジョンが結果に含まれなければならない場合に指定されます。true の場合、エンティティーのリビジョンタイプが **DEL** になり、id 以外のすべてのフィールドの値が **null** になります。

例11.35 特定のプロパティーを修正したエンティティーのクエリリビジョン

下記のクエリは、**actualDate** プロパティーが変更された、指定の ID を持つ **MyEntity** のすべてのリビジョンを返します。

```
AuditQuery query = getAuditReader().createQuery()
  .forRevisionsOfEntity(MyEntity.class, false, true)
  .add(AuditEntity.id().eq(id));
  .add(AuditEntity.property("actualDate").hasChanged())
```

hasChanged 条件を他の基準と組み合わせることができます。次のクエリは、*revisionNumber* 生成時に **MyEntity** の水平スライスを返します。これは、**prop1** は修正され、**prop2** は修正されなかったリビジョンに限定されます。

```
AuditQuery query = getAuditReader().createQuery()
  .forEntitiesAtRevision(MyEntity.class, revisionNumber)
  .add(AuditEntity.property("prop1").hasChanged())
  .add(AuditEntity.property("prop2").hasNotChanged());
```

結果セットには *revisionNumber* よりも小さい番号のリビジョンも含まれます。これは、このクエリが「**prop1** は修正され、**prop2** はそのままの *revisionNumber* で変更された **MyEntities** をすべて返す」とは読み取られないことを意味します。

次のクエリは **forEntitiesModifiedAtRevision** を使用してこの結果がどのように返されるか表しています。

```
AuditQuery query = getAuditReader().createQuery()
  .forEntitiesModifiedAtRevision(MyEntity.class, revisionNumber)
  .add(AuditEntity.property("prop1").hasChanged())
  .add(AuditEntity.property("prop2").hasNotChanged());
```

例11.36 特定のリビジョンで修正されたクエリエンティティー

次の例は、特定のリビジョンで変更されたエンティティーに対する基本的なクエリになります。読み出される特定のリビジョンで、エンティティー名と対応する Java クラスを変更できます。

```
Set<Pair<String, Class>> modifiedEntityTypes = getAuditReader()

.getCrossTypeRevisionChangesReader().findEntityTypes(revisionNumber);
```

org.hibernate.envers.CrossTypeRevisionChangesReader からもアクセスできる他のクエリーは以下のとおりです。

List<Object> findEntities(Number)

特定のリビジョンで変更 (追加、更新、削除）されたすべての監査されたエンティティーのスナップショットを返します。**n+1** 個の SQL クエリを実行します (**n** は指定のリビジョン内で変更された異なるエンティティークラスの数になります)。

List<Object> findEntities(Number, RevisionType)

変更タイプによってフィルターされた特定のリビジョンで変更 (追加、更新、削除) されたすべての監査されたエンティティーのスナップショットを返します。**n+1** 個の SQL クエリを実行します (**n** は指定のリビジョン内で変更された異なるエンティティークラスの数になります)。

`Map<RevisionType, List<Object>> findEntitiesGroupByRevisionType(Number)`

修正操作 (追加、更新、削除など) によってグループ化されたエンティティースナップショットの一覧が含まれるマップを返します。**3n+1** 個の SQL クエリを実行します (**n** は指定のリビジョン内で変更された異なるエンティティークラスの数になります)。

バグを報告する

第12章 JAX-RS WEB サービス

12.1. JAX-RS について

JAX-RS は RESTful Web サービス向けの Java API です。JAX-RS は、REST を利用しアノテーションを使うことで Web サービスの構築をサポートします。このようなアノテーションにより、 Java オブジェクトを Web リソースにマッピングするプロセスが簡素化されます。JAX-RS の仕様は http://www.jcp.org/en/jsr/detail?id=311 で定義されています。

RESTEasy は JAX-RS の JBoss Enterprise Application Platform 6 実装で、この仕様に対し機能を追加しています。

JBoss Enterprise Application Platform 6 は JSR 311 - JAX-RS に完全準拠しています。

JPA および JBoss Enterprise Application Platform 6 を利用するには、**helloworld-rs**、**jax-rs-client**、**kitchensink** クイックスタート: 「Java EE クイックスタートサンプルへのアクセス」を参照してくだい。

バグを報告する

12.2. RESTEASY について

RESTEasy は JAX-RS Java API の移植可能な実装で、リモートサーバーへの送信要求をマッピングするためのクライアントサイドフレームワーク (RESTEasy JAX-RS クライアントフレームワーク) を含む追加機能も提供し JAX-RS がクライアントあるいはサーバー側の仕様として動作するようにします。

バグを報告する

12.3. RESTFUL WEB サービスについて

RESTful Web サービスは、API を Web に公開するために設計されています。クライアントが予測可能な URL を使うことでデーターやリソースへアクセスできるようにし、従来の Web サービスよりもパフォーマンス、拡張性、柔軟性の向上を目指しています。

RESTful サービス の Java Enterprise Edition 6 仕様は JAX-RS で、JAX-RS に関する詳細情報は、「JAX-RS について」および 「RESTEasy について」 を参照してください。

バグを報告する

12.4. RESTEASY 定義済みアノテーション

表12.1 JAX-RS/RESTEasy アノテーション

アノテーション	使用法
ClientResponseType	これは、Response の戻り値タイプを持つ RESTEasy クライアントインターフェースに追加できるアノテーションです。
ContentEncoding	アノテートされたアノテーションを使用して適用する Content-Encoding を指定するメタアノテーション。

アノテーション	使用法
DecorateTypes	サポートされたタイプを指定するために DecoratorProcessor クラスに配置する必要があります。
Decorator	デコレーションをトリガーする別のアノテーションに配置するメタアノテーション。
Form	これは、受信/送信の要求/応答用の値オブジェクトとして使用できます。
StringParameterUnmarshallerBinder	文字列ベースのアノテーションインジェクターに適用する StringParameterUnmarshaller をトリガーする別のアノテーションに配置するメタアノテーション。
Cache	応答の Cache-Control ヘッダーを自動的に設定します。
NoCache	Cache-Control 応答ヘッダーを "nocache" に設定します。
ServerCached	この jax-rs メソッドへの応答をサーバーでキャッシュするよう指定します。
ClientInterceptor	インターセプターをクライアントサイドインターセプターとして識別します。
DecoderPrecedence	このインターセプターは、Content-Encoding デコーダーです。
EncoderPrecedence	このインターセプターは、Content-Encoding エンコーダーです。
HeaderDecoratorPrecedence	HeaderDecoratorPrecedence インターセプターは、応答 (サーバー上) または送信要求 (クライアント上) を特別なユーザー定義ヘッダーでデコレートするため、常に最初に使用する必要があります。
RedirectPrecedence	PreProcessInterceptor に配置する必要があります。
SecurityPrecedence	PreProcessInterceptor に配置する必要があります。
ServerInterceptor	インターセプターをサーバーサイドインターセプターとして識別します。

アノテーション	使用法
NoJackson	Jackson プロバイダーをトリガーしない場合にクラス、パラメーター、フィールド、またはメソッドに配置されます。
ImageWriterParams	IIOImageProvider にパラメーターを渡すためにリソースクラスが使用できるアノテーション。
DoNotUseJAXBProvider	JAXB MessageBodyReader/Writer を使用せず、タイプをマーシャルするためにさらに特定のプロバイダーを使用する場合は、これをクラスまたはパラメーターに配置します。
Formatted	XML 出力をインデントと改行を使用して書式設定します。これは JAXB デコレーターです。
IgnoreMediaTypes	特定のメディタイプに JAXB プロバイダーを使用しないよう JAXRS に指示するために、タイプ、メソッド、パラメーター、またはフィールドに配置されます。
Stylesheet	XML スタイルシートヘッダーを指定します。
Wrapped	JAXB オブジェクトのコレクションまたはアレイをマーシャルまたはマーシャル解除する場合は、これをメソッドまたはパラメーターに配置します。
WrappedMap	JAXB オブジェクトのマップをマーシャルまたはマーシャル解除する場合は、これをメソッドまたはパラメーターに配置します。
XmlHeader	返されたドキュメントの XML ヘッダーを設定します。
BadgerFish	JSONConfig
Mapped	JSONConfig
XmlNsMap	JSONToXml
MultipartForm	これは、multipart/form-data mime タイプの受信/送信の要求/応答用の値オブジェクトとして使用できます。
PartType	リストまたはマップを multipart/* タイプを書きだす場合に、Multipart プロバイダーとともに使用する必要があります。

アノテーション	使用法
XopWithMultipartRelated	このアノテーションは、JAXB アノテートオブジェクトに対して受信/送信 XOP メッセージ (multipart/related としてパッケージ化) を処理/生成するために使用できます。
After	署名または古さの検証を行う場合に、期限切れ属性を追加するために使用されます。
Signed	DOSETA 仕様を使用して要求または応答の署名をトリガーする便利なアノテーション。
Verify	署名ヘッダーに指定された入力署名の検証。

バグを報告する

12.5. RESTEASY 設定

12.5.1. RESTEasy 設定パラメーター

表12.2 要素

オプション名	デフォルト値	説明
resteasy.servlet.mapping.prefix	デフォルトなし	Resteasy servlet-mapping の url-pattern が /* でない場合
resteasy.scan	false	WEB-INF/lib jar および WEB-INF/classes ディレクトリを自動的にスキャンして @Provider と JAX-RS の両方のリソースクラス (@Path、@GET、@POST など) を探し、それらを登録します。
resteasy.scan.providers	false	@Provider クラスをスキャンし、それらを登録します。
resteasy.scan.resources	false	JAX-RS リソースクラスをスキャンします。
resteasy.providers	デフォルトなし	登録する完全修飾 @Provider クラス名のコンマ区切りのリスト
resteasy.use.builtin.providers	true	デフォルトを登録するか否かに関わらず、ビルトイン @Provider クラス

オプション名	デフォルト値	説明
resteasy.resources	デフォルトなし	登録する完全修飾 JAX-RS リソースクラス名のコンマ区切りのリスト
resteasy.jndi.resources	デフォルトなし	JAX-RS リソースとして登録するオブジェクトを参照する JNDI 名のコンマ区切りのリスト
javax.ws.rs.Application	デフォルトなし	仕様の移植可能な方法でブートストラップを行うアプリケーションクラスの完全修飾名
resteasy.media.type.mappings	デフォルトなし	ファイル名の拡張子 (例: .xml、.txt など) をメディアタイプにマッピングすることにより、Accept ヘッダーが必要なくなります。クライアントで Accept ヘッダーを使用して表現を選択することができない場合に使用します (ブラウザーなど)。
resteasy.language.mappings	デフォルトなし	ファイル名の拡張子 (例: .en、.fr など) を言語にマッピングすることにより、Accept-Language ヘッダーの必要がなくなります。クライアントで Accept-Language ヘッダーを使用して言語を選択することができない場合に使用します (ブラウザーなど)。

重要

Servlet 3.0 コンテナで、`web.xml` ファイル内の `resteasy.scan.*` 設定が無視され、すべての JAX-RS 自動コンポーネントが自動的にスキャンされます。

バグを報告する

12.6. JAX-RS WEB サービスセキュリティー

12.6.1. RESTEasy JAX-RS Web サービスのロールベースのセキュリティーを有効にする

概要

RESTEasy は JAX-RS メソッドの @RolesAllowed、@PermitAll、@DenyAll アノテーションをサポートしますが、デフォルトではこれらのアノテーションを認識しません。次の手順に従って `web.xml` ファイルを設定し、ロールベースセキュリティーを有効にします。

警告

アプリケーションが EJB を使用する場合はロールベースセキュリティーを有効にしないでください。RESTEasy ではなく EJB コンテナが機能を提供します。

手順12.1 RESTEasy JAX-RS Web サービスのロールベースのセキュリティーを有効にする

1. テキストエディターでアプリケーションの **web.xml** ファイルを開きます。
2. 以下の **<context-param>** をファイルの **web-app** タグ内に追加します。

```
<context-param>
    <param-name>resteasy.role.based.security</param-name>
    <param-value>true</param-value>
</context-param>
```

3. **<security-role>** タグを使用して RESTEasy JAX-RS WAR ファイル内で使用されるすべてのロールを宣言します。

```
<security-role><role-name>ROLE_NAME</role-name></security-role>
<security-role><role-name>ROLE_NAME</role-name></security-role>
```

4. すべてのロールに対して JAX-RS ランタイムが対応する全 URL へのアクセスを承認します。

```
<security-constraint><web-resource-collection><web-resource-
name>Resteasy</web-resource-name><url-pattern>/PATH</url-pattern>
</web-resource-collection><auth-constraint><role-
name>ROLE_NAME</role-name><role-name>ROLE_NAME</role-name></auth-
constraint></security-constraint>
```

結果

ロールベースセキュリティーが定義されたロールのセットによりアプリケーション内で有効になります。

例12.1 ロールベースセキュリティーの設定例

```
<web-app>

   <context-param>
<param-name>resteasy.role.based.security</param-name>
<param-value>true</param-value>
   </context-param>

   <servlet-mapping>
<servlet-name>Resteasy</servlet-name>
<url-pattern>/*</url-pattern>
   </servlet-mapping>

   <security-constraint>
<web-resource-collection>
    <web-resource-name>Resteasy</web-resource-name>
    <url-pattern>/security</url-pattern>
</web-resource-collection>
<auth-constraint>
    <role-name>admin</role-name>
    <role-name>user</role-name>
</auth-constraint>
   </security-constraint>

   <security-role>
<role-name>admin</role-name>
   </security-role>
   <security-role>
<role-name>user</role-name>
   </security-role>

</web-app>
```

バグを報告する

12.6.2. アノテーションを使用した JAX-RS Web サービスの保護

概要

サポート対象のセキュリティーアノテーションを使用して JAX-RS Web サービスを保護する手順を取り上げます。

手順12.2 サポート対象のセキュリティーアノテーションを使用した JAX-RS Web サービスのセキュア化

1. ロールベースセキュリティーを有効にします。詳細は 「RESTEasy JAX-RS Web サービスのロールベースのセキュリティーを有効にする」 を参照してください。
2. JAX-RS Web サービスにセキュリティーアノテーションを追加します。RESTEasy は次のアノテーションをサポートします。

   @RolesAllowed

   メソッドにアクセスできるロールを定義します。ロールはすべて **web.xml** ファイルに定義する必要があります。

@PermitAll

`web.xml` ファイルに定義されている全ロールによるメソッドへのアクセスを許可します。

@DenyAll

メソッドへのアクセスをすべて拒否します。

バグを報告する

12.7. RESTEASY ロギング

12.7.1. JAX-RS Web サービスロギングについて

RESTEasy は、java.util.logging、log4j、および slf4j 経由でロギングをサポートします。フレームワークは次のアルゴリズムより選択されます。

1. log4j がアプリケーションのクラスパスにある場合は、log4j が使用されます。
2. slf4j がアプリケーションのクラスパスにある場合は、slf4j が使用されます。
3. log4j と slf4j がいずれもクラスパスにない場合は、java.util.logging がデフォルトで使用されます。
4. サーブレットのコンテキストパラメーター resteasy.logger.type が java.util.logging、log4j、または slf4j に設定された場合は、このデフォルトの動作がオーバーライドされます。

JAX-RS アプリケーションのロギングを設定するには、「管理コンソールのログカテゴリーの設定」を参照してください。

バグを報告する

12.7.2. 管理コンソールのログカテゴリーの設定

ログカテゴリーは、どのログメッセージをキャプチャーし、どのログハンドラーに送信するかを定義します。

重要

現在、ログカテゴリーはサーバー設定ファイルのみに完全設定することができます。最終リリースでは、管理コンソールやコマンドライン管理ツールを使用した完全設定がサポートされる予定です。

サーバー設定ファイルに新しいログハンドラーを追加するには、次の手順を実行します。

1. **サーバーの停止**
   JBoss Enterprise Application Platform 6 のサーバーが稼動している場合は停止します。
2. **テキストエディターで設定ファイルを開く**
   Enterprise Application Platform を管理ドメインまたはスタンドアローンサーバーで稼動しているかどうかによって、デフォルトの設定ファイルの場所が異なります。管理ドメインの場合は ***EAP_HOME*/domain/configuration/domain.xml**、スタンドアローンサーバーの場合は ***EAP_HOME*/standalone/configuration/standalone.xml** になります。

3. **ロギングサブシステムノードの検索**
   ロギングサブシステム設定ノードを見つけます。ロギングサブシステム設定ノードは次のようになります。

```
<subsystem xmlns="urn:jboss:domain:logging:1.1">

</subsystem>
```

   ロギングサブシステム設定には、すでにログハンドラーやカテゴリーなどの多くの項目が含まれています。

4. **新規ロガーノードの追加**
   新しい **<logger>** ノードをロギングシステムノードの子として追加します。このロガーがメッセージを受信するクラス名またはパッケージ名である **category** という属性を持たなければなりません。

```
<logger category="com.acme.accounts.receivables">

</logger>
```

   任意で **use-parent-handlers** 属性を追加することもできます。この属性は**true** か **false** に設定できます。**true** に設定すると、このログカテゴリーによって受信されたすべてのメッセージもルートロガーのハンドラーによって処理されます。指定がない場合、**true** がデフォルトとして適用されます。

5. **ログレベルの指定**
   name という名前の属性を持つ **<level>** 要素を追加します。name の値は、このカテゴリーが適用するログレベルである必要がががあります。

```
<logger category="com.acme.accounts.receivables">
   <level name="DEBUG"/>
</logger>
```

6. **任意の設定: ログハンドラーの指定**
   このカテゴリーからのログメッセージを処理するため使用したい各ログハンドラーに対して、**<handler>** 要素が含まれる **<handlers>** 要素を追加します。

   指定されているハンドラーがなく、use-parent-handler が true に設定されていない場合、これ以上ログメッセージは処理されません。指定されているハンドラーがなくても use-parent-handler が true に設定されている場合は root-logger のハンドラーが使用されます。

```
<logger category="com.acme.accounts.receivables">
   <level name="DEBUG"/>
   <handlers>
      <handler name="ACCOUNTS-DEBUG-FILE"/>
   </handlers>
</logger>
```

7. JBoss Enterprise Application Platform 6 サーバーを再起動します。

バグを報告する

12.7.3. RESTEasy で定義されたロギングカテゴリー

表12.3 カテゴリー

カテゴリー	関数
org.jboss.resteasy.core	コア RESTEasy 実装により全アクティビティをログに記録します。
org.jboss.resteasy.plugins.providers	RESTEasy エンティティープロバイダーにより全アクティビティをログに記録します。
org.jboss.resteasy.plugins.server	RESTEasy サーバー実装により全アクティビティをログに記録します。
org.jboss.resteasy.specimpl	JAX-RS 実装クラスにより全アクティビティをログに記録します。
org.jboss.resteasy.mock	RESTEasy モックフレームワークにより全アクティビティをログに記録します。

バグを報告する

12.8. 例外処理

12.8.1. 例外マッパーの作成

概要

例外マッパーはスローされた例外をキャッチし、特定の HTTP 応答を書き込むコンポーネントで、アプリケーションによって提供されます。

> **例12.2 例外マッパー**
>
> 例外マッパーは @Provider アノテーションがアノテートされるクラスであり、**ExceptionMapper** インターフェースを実装します。
>
> 例外マッパーの例は次の通りです。
>
> ```
> @Provider
> public class EJBExceptionMapper implements
> ExceptionMapper<javax.ejb.EJBException>
>     {
>     Response toResponse(EJBException exception) {
>     return Response.status(500).build();
>     }
> }
> ```

例外マッパーを登録するには **resteasy.providers** コンテキストパラメーター下の**web.xml** に例外マッパーをリストするか、プログラムを使用して **ResteasyProviderFactory** クラスより例外マッパーを登録します。

バグを報告する

12.8.2. RESTEasy 内部で送出された例外

表12.4 例外リスト

例外	HTTP コード	説明
BadRequestException	400	不正なリクエスト。このリクエストは正しくフォーマットされていないか、リクエスト入力の処理に問題があります。
UnauthorizedException	401	権限がありません。RESTEasy のアノテーションベース、ロールベースのセキュリティを利用している場合セキュリティの例外が出されます。
InternalServerErrorException	500	内部サーバーエラー
MethodNotAllowedException	405	呼び出した HTTP オペレーションを処理できるリソースに、JAX-RS メソッドがありません。
NotAcceptableException	406	Accept ヘッダーに記載されているメディアタイプを生成できる JAX-RS がありません。
NotFoundException	404	リクエストパス／リソースにサービスを提供する JAX-RS メソッドがありません。
ReaderException	400	**`MessageBodyReaders`** からスローされた例外はすべて、この例外の中にラップされます。ラップされた例外に **`ExceptionMapper`** がないか、あるいは例外が **`WebApplicationException`** でない場合、RESTEasy はデフォルトで 400 コードを返します。
WriterException	500	**`MessageBodyWriters`** からスローされた例外はすべて、この例外の中にラップされます。ラップされた例外に **`ExceptionMapper`** がないか、あるいは例外が **`WebApplicationException`** でない場合、RESTEasy はデフォルトで 400 コードを返します。

例外	HTTP コード	説明
o.j.r.plugins.providers.jaxb.JAXBUnmarshalException	400	JAXB プロバイダー (XML および Jettison) はこの例外を読み込み時にスローします。JAXBExceptions をラッピングする場合もあります。このクラスは **`ReaderException`** を拡張します。
o.j.r.plugins.providers.jaxb.JAXBMarshalException	500	JAXB プロバイダー (XML および Jettison) はこの例外を書き込み時にスローします。JAXBExceptions をラッピングする場合もあります。このクラスは **`WriterException`** を拡張します。
ApplicationException	N/A	アプリケーションコードからスローされた例外をすべてラップします。**`InvocationTargetException`** と同じように機能します。ラップされた例外に ExceptionMapper がある場合は、要求を処理するために使用されます。
Failure	N/A	内部 RESTEasy エラー。ログに記録されません。
LoggableFailure	N/A	内部 RESTEasy エラー。ログに記録されています。
DefaultOptionsMethodException	N/A	ユーザーが **`HTTP OPTIONS`** を呼び出し HTTP OPTIONS に対する JAX-RS メソッドがない場合、RESTEasy ではデフォルトで例外をスローします。

バグを報告する

12.9. RESTEASY インターセプター

12.9.1. JAX-RS 呼び出しのインターセプト

概要

RESTEasy は JAX-RS 呼び出しをインターセプトでき、インターセプターと呼ばれるリスナーのようなオブジェクトでルーティングを行います。このトピックでは、インターセプター4種について説明していきます。

■

例12.3 MessageBodyReader/Writer インターセプター

MessageBodyReaderInterceptors および MessageBodyWriterInterceptors をサーバー側、クライアント側いずれでも利用可能です。RESTEasy がインターセプター一覧に追加するか否かがわかるように **@Provider** と、 **@ServerInterceptor** または **@ClientInterceptor** がアノテートされます。

これらのインターセプターは、**MessageBodyReader.readFrom()** あるいは **MessageBodyWriter.writeTo()** の呼び出しをラップします。また、出力、入力ストリームをラップする際にも利用可能です。

RESTEasy GZIP サポートには、Gzip エンコーディングが機能できるよう、デフォルトの出力、入力ストリームを GzipOutputStream あるいは GzipInputStream で作成しオーバーライドするインターセプターがあります。また、これらのインターセプターを使い、ヘッダーにレスポンスを追加、あるいはクライアント側の送信リクエストを追加できます。

```
public interface MessageBodyReaderInterceptor
  {
     Object read(MessageBodyReaderContext context) throws IOException,
WebApplicationException;

  }

public interface MessageBodyWriterInterceptor
  {
     void write(MessageBodyWriterContext context) throws IOException,
WebApplicationException;

  }
```

インターセプターおよび MessageBodyReader／Writer が大きな Java の呼び出しスタックで呼び出されます。次のインターセプターに移動するには、**MessageBodyReaderContext.proceed()** あるいは **MessageBodyWriterContext.proceed()** が呼び出され、これ以上呼び出すインターセプターがない場合、MessageBodyReader あるいは MessageBodyWriter の **readFrom()** あるいは **writeTo()** が呼び出されます。このラッピングにより、オブジェクトがReader あるいは Writer に到達前にオブジェクトを変更することができ、**proceed()** を返すまでに消去できます。

以下の例はサーバー側のインターセプターで、ヘッダーの値をレスポンスに追加します。

```
@Provider
@ServerInterceptor
public class MyHeaderDecorator implements MessageBodyWriterInterceptor {

    public void write(MessageBodyWriterContext context) throws
IOException, WebApplicationException
    {
       context.getHeaders().add("My-Header", "custom");
       context.proceed();
    }
}
```

例12.4 PreProcessInterceptor

呼び出しが行えるように JAX-RS リソースメソッドを検出してから実際に呼び出す前に、PreProcessInterceptors が実行されます。@ServerInterceptor でアノテーションが付けられ、順番に実行されます。

これらのインターフェースはサーバー上でのみ利用可能です。このインターフェースを使いセキュリティ機能を実装するか、Java リクエストを処理します。RESTEasy セキュリティ実装は、ユーザーが認証情報を渡さない場合に実際に操作が行われる前にインターセプターのこの型を使いリクエストを中断します。RESTEasy のキャッシュフレームワークはこれを使い、キャッシュされたレスポンスを返し、メソッドが再度呼び出されないようにします。

```
public interface PreProcessInterceptor
    {
       ServerResponse preProcess(HttpRequest request, ResourceMethod
method) throws Failure, WebApplicationException;
    }
```

preProcess() メソッドは ServerResponse を返し、基礎となる JAX-RS メソッドは呼び出されず、ランタイムがレスポンスを処理しクライアントに返します。**preProcess()** メソッドが ServerResponse を返さない場合は、基礎となる JAX-RS メソッドが呼び出されます。

例12.5 PostProcessInterceptors

PostProcessInterceptors は、MessageBodyWriters の呼び出し前かつ JAX-RS メソッドが呼び出された後に実行されます。MessageBodyWriter が呼び出されずレスポンスヘッダーが設定された場合に利用されます。

サーバー側でのみ利用可能です。何もラップしませんが、順番に呼び出されます。

```
public interface PostProcessInterceptor
  {
    void postProcess(ServerResponse response);
  }
```

例12.6 ClientExecutionInterceptors

ClientExecutionInterceptors はクライアント側でのみ利用可能です。サーバーに対する HTTP 呼び出し関連でラップを行います。これらは、**@ClientInterceptor** や **@Provider** のアノテーションが付けられます。MessageBodyWriter の実行後、および ClientRequest がクライアント側で構築された後に実行されます。

RESTEasy GZIP サポートは、ClientExecutionInterceptors を使いリクエストが出される前に Accept ヘッダーに "gzip, deflate" を含めるよう設定します。RESTEasy のクライアントキャッシュはこれを使い、キャッシュにリソースが含まれているか確認します。

```
public interface ClientExecutionInterceptor
{
  ClientResponse execute(ClientExecutionContext ctx) throws Exception;
}
```

```
public interface ClientExecutionContext
{
  ClientRequest getRequest();

  ClientResponse proceed() throws Exception;
}
```

バグを報告する

12.9.2. インターセプターを JAX-RS メソッドにバインド

概要

デフォルトでは、登録済みの全インターセプターが、全リクエストに対して呼び出されます。**AcceptedByMethod** インターフェースを実装しこの動作を調整することができます。

例12.7 インターセプターの例を構築

RESTEasy は **AcceptedByMethod** インターフェースを実装するインターセプターに対して **accept()** メソッドを呼び出します。このメソッドが true を返す場合、インターセプターが JAX-RS メソッドの呼び出しチェーンに追加され、True が返されない場合はそのメソッドについては無視されます。

以下の例では、 **accept()** が @GET アノテーションが JAX-RS メソッドに存在するかを判断します。存在する場合、インターセプターがこのメソッドの呼び出しチェーンに適用されます。

```
@Provider
@ServerInterceptor
public class MyHeaderDecorator implements MessageBodyWriterInterceptor,
AcceptedByMethod {

    public boolean accept(Class declaring, Method method) {
       return method.isAnnotationPresent(GET.class);
    }

   public void write(MessageBodyWriterContext context) throws
IOException, WebApplicationException
   {
      context.getHeaders().add("My-Header", "custom");
      context.proceed();
   }
}
```

バグを報告する

12.9.3. インターセプターの登録

概要

このトピックでは、RESTEasy JAX-RS インターセプターをアプリケーションに登録する方法を説明しています。

手順12.3 インターセプターの登録

- インターセプターを登録するには、**resteasy.providers** context-param の **web.xml** ファイルにインターセプターをリストアップし、**Application.getClasses()** あるいは **Application.getSingletons()** メソッドにてクラスあるいはオブジェクトとして返します。

バグを報告する

12.9.4. インターセプター優先度ファミリー

12.9.4.1. インターセプター優先度ファミリーについて

概要

インターセプターは呼び出される順番に敏感なことがあります。RESTEasy はインターセプターをファミリーにグループ化し、順番付けを簡易化します。この参照トピックではビルトインのインターセプター優先度ファミリーと各ファミリーに関連付けられるインターセプターについて取り上げます。

事前定義されたファミリーは 5 つあります。これらのファミリーは次の順序で呼び出されます。

SECURITY

SECURITY インターセプターは通常 PreProcessInterceptors です。呼び出しが承認されるまでの時間を最低限にするためこのインターセプターが最初に呼び出されます。

HEADER_DECORATOR

HEADER_DECORATOR インターセプターは応答または送信要求へヘッダーを追加します。追加されたヘッダーが他のインターセプターファミリーの挙動に影響することがあるため、SECURITY インターセプターの後に呼び出されます。

ENCODER

ENCODER インターセプターは OutputStream を変更します。GZIP インターセプターは GZIPOutputStream を作成し、圧縮するため真の OutputStream をラッピングします。

REDIRECT

REDIRECT インターセプターは要求のルートを変更し、JAX-RS メソッドを完全に迂回することがあるため、通常 PreProcessInterceptors で使用されます。

DECODER

DECODER インターセプターは inputStream をラッピングします。例えば、GZIP インターセプターデコーダーは GzipInputStream インスタンスの InputStream をラッピングします。

優先度ファミリーに関連付けられていないインターセプターは最後に呼び出されます。インターセプターに優先度ファミリーを割り当てるには、「RESTEasy 定義済みアノテーション」 の通り @Precedence アノテーションを使用します。

バグを報告する

12.9.4.2. カスタムのインターセプター優先グループ (Precedence Family) を定義

概要

カスタムの Precedence Family は**web.xml** ファイルで作成、登録できます。このトピックでは、インターセプターの Precedence Family を定義する際に利用可能なコンテキストパラメーターの例をみていきます。

新規の Precedence Family を定義する際に利用可能なコンテキストパラメーターは 3 種類あります。

例12.8 resteasy.append.interceptor.precedence

resteasy.append.interceptor.precedence のコンテキストパラメーターは、デフォルトの precedence family 一覧に新規の family を追加します。

```
<context-param>
    <param-name>resteasy.append.interceptor.precedence</param-name>
    <param-value>CUSTOM_PRECEDENCE_FAMILY</param-value>
</context-param>
```

例12.9 resteasy.interceptor.before.precedence

resteasy.interceptor.before.precedence コンテキストパラメーターは、カスタムの Family が先に実行されるようにデフォルトの Precedence Family を定義します。 このパラメーターの値は、 *DEFAULT_PRECEDENCE_FAMILY/CUSTOM_PRECEDENCE_FAMILY*を **:** で区切るという形式をとります。

```
<context-param>
    <param-name>resteasy.interceptor.before.precedence</param-name>
    <param-value>DEFAULT_PRECEDENCE_FAMILY :
CUSTOM_PRECEDENCE_FAMILY</param-value>
</context-param>
```

例12.10 resteasy.interceptor.after.precedence

resteasy.interceptor.after.precedence コンテキストパラメーターは、カスタムの Family が後で実行されるようにデフォルトの Precedence Family を定義します。 このパラメーターの値は、 *DEFAULT_PRECEDENCE_FAMILY/CUSTOM_PRECEDENCE_FAMILY*を **:** で区切るという形式をとります。

```
<context-param>
    <param-name>resteasy.interceptor.after.precedence</param-name>
    <param-value>DEFAULT_PRECEDENCE_FAMILY :
CUSTOM_PRECEDENCE_FAMILY</param-value>
</context-param>
```

Precedence Family は @Precedence アノテーションを使いインターセプターに適用されます。デフォルトの Precedence Family 一覧については、「インターセプター優先度ファミリーについて」を参照してください。

バグを報告する

12.10. 文字列ベースのアノテーション

12.10.1. 文字列ベースの @*Param Annotations をオブジェクトに変換

@PathParam や @FormParamJAX-RS などの **@*Param** アノテーションは、Raw HTTP リクエストの文字列として表現されます。これらのオブジェクトに**valueOf(String)** の静的メソッドあるいは String パラメーターを取るコンストラクターが含まれている場合、注入されたパラメーターの型をオブジェクトに変換することが可能です。

RESTEasy は専用の **@Provider** インターフェースを2つ提供し、**valueOf(String)** の静的メソッドあるいは文字列コンストラクターのいずれも持たないクラスに対して、この変換処理を行います。

例12.11 StringConverter

StringConverter インターフェースは、カスタム文字列のマーシャリングが行えるよう実装されています。このインターフェースは web.xml ファイルの resteasy.providers context-param 下で登録されています。また、ResteasyProviderFactory.addStringConverter() メソッドを呼び出すことにより手動で登録することもできます。

以下の例は、簡単な StringConverter の使用例です。

```
import org.jboss.resteasy.client.ProxyFactory;
import org.jboss.resteasy.spi.StringConverter;
import org.jboss.resteasy.test.BaseResourceTest;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import javax.ws.rs.HeaderParam;
import javax.ws.rs.MatrixParam;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.ext.Provider;

public class StringConverterTest extends BaseResourceTest
{
  public static class POJO
  {
     private String name;

     public String getName()
     {
 return name;
     }

     public void setName(String name)
     {
 this.name = name;
     }
  }

  @Provider
  public static class POJOConverter implements StringConverter<POJO>
```

```
  {
     public POJO fromString(String str)
     {
 System.out.println("FROM STRNG: " + str);
 POJO pojo = new POJO();
 pojo.setName(str);
 return pojo;
     }

     public String toString(POJO value)
     {
 return value.getName();
     }
  }

  @Path("/")
  public static class MyResource
  {
     @Path("{pojo}")
     @PUT
     public void put(@QueryParam("pojo")POJO q, @PathParam("pojo")POJO
pp,
       @MatrixParam("pojo")POJO mp, @HeaderParam("pojo")POJO hp)
     {
 Assert.assertEquals(q.getName(), "pojo");
 Assert.assertEquals(pp.getName(), "pojo");
 Assert.assertEquals(mp.getName(), "pojo");
 Assert.assertEquals(hp.getName(), "pojo");
     }
  }

  @Before
  public void setUp() throws Exception
  {

dispatcher.getProviderFactory().addStringConverter(POJOConverter.class);
     dispatcher.getRegistry().addPerRequestResource(MyResource.class);
  }

  @Path("/")
  public static interface MyClient
  {
     @Path("{pojo}")
     @PUT
     void put(@QueryParam("pojo")POJO q, @PathParam("pojo")POJO pp,
       @MatrixParam("pojo")POJO mp, @HeaderParam("pojo")POJO hp);
  }

  @Test
  public void testIt() throws Exception
  {
     MyClient client = ProxyFactory.create(MyClient.class,
"http://localhost:8081");
     POJO pojo = new POJO();
     pojo.setName("pojo");
     client.put(pojo, pojo, pojo, pojo);
```

```
   }
}
```

例12.12 StringParameterUnmarshaller

StringParameterUnmarshaller インターフェースは、パラメーターに付けられたアノテーションや、注入先のフィールドからの影響を受けます。これは、インジェクターごとに作成されます。**setAnnotations()** メソッドは、**resteasy** に呼び出されアンマーシャラーを初期化します。

インターフェースを実装するプロバイダーを作成、登録することで、このインターフェースを追加できます。また、**org.jboss.resteasy.annotations.StringsParameterUnmarshallerBinder** と呼ばれる **meta-annotation** を使いバインドすることもできます。

以下の例では、**java.util.Date** ベースの **@PathParam** をフォーマットします。

```
public class StringParamUnmarshallerTest extends BaseResourceTest
{
   @Retention(RetentionPolicy.RUNTIME)
   @StringParameterUnmarshallerBinder(DateFormatter.class)
   public @interface DateFormat
   {
      String value();
   }

   public static class DateFormatter implements
StringParameterUnmarshaller<Date>
   {
      private SimpleDateFormat formatter;

      public void setAnnotations(Annotation[] annotations)
      {
         DateFormat format = FindAnnotation.findAnnotation(annotations,
DateFormat.class);
         formatter = new SimpleDateFormat(format.value());
      }

      public Date fromString(String str)
      {
         try
         {
            return formatter.parse(str);
         }
         catch (ParseException e)
         {
            throw new RuntimeException(e);
         }
      }
   }

   @Path("/datetest")
   public static class Service
   {
```

```
      @GET
      @Produces("text/plain")
      @Path("/{date}")
      public String get(@PathParam("date") @DateFormat("MM-dd-yyyy")
Date date)
      {
         System.out.println(date);
         Calendar c = Calendar.getInstance();
         c.setTime(date);
         Assert.assertEquals(3, c.get(Calendar.MONTH));
         Assert.assertEquals(23, c.get(Calendar.DAY_OF_MONTH));
         Assert.assertEquals(1977, c.get(Calendar.YEAR));
         return date.toString();
      }
   }

   @BeforeClass
   public static void setup() throws Exception
   {
      addPerRequestResource(Service.class);
   }

   @Test
   public void testMe() throws Exception
   {
      ClientRequest request = new
ClientRequest(generateURL("/datetest/04-23-1977"));
      System.out.println(request.getTarget(String.class));
   }
}
```

@DateFormat と呼ばれる新しいアノテーションを定義します。このアノテーションは、**DateFormater** クラスへの参照を持つ **meta-annotation StringParameterUnmarshallerBinder** で注釈されます。

Service.get() メソッドには**@DateFormat** のアノテーションもついた **@PathParam** パラメーターがあります。**@DateFormat** を適用すると、**DateFormatter** のバインディングがトリガーされます。すると、**DateFormatter** が実行され、**get()** メソッドの **date** パラメーターへ **path** パラメーターをアンマーシャリングします。

バグを報告する

12.11. ファイル拡張子の設定

12.11.1. web.xml ファイルでメディアタイプへファイル拡張子をマッピングする

概要

クライアントによっては、ブラウザー同様に表現のメディアタイプや言語のネゴシエーションに **Accept** および **Accept-Language** ヘッダーを使用できないものがあります。**RESTEasy** はこの問題に対処するため、ファイル名のサフィックスをメディアタイプや言語にマッピングすることができます。次の手順に従って、**web.xml** ファイルにてメディアタイプをファイル拡張子にマッピングします。

手順12.4 メディアタイプとファイル拡張子のマッピング

1. テキストエディターでアプリケーションの **web.xml** ファイルを開きます。
2. コンテキストパラメーター **resteasy.media.type.mappings** をファイルの **web-app** タグ内に追加します。

```
<context-param>
    <param-name>resteasy.media.type.mappings</param-name>
</context-param>
```

3. パラメーター値を設定します。マッピングはコンマ区切りリストを形成します。各マッピングは：で区切られます。

例12.13 マッピングの例

```
<context-param>
    <param-name>resteasy.media.type.mappings</param-name>
    <param-value>html : text/html, json : application/json, xml :
application/xml</param-name>
</context-param>
```

バグを報告する

12.11.2. web.xml ファイルにてファイル拡張子を言語にマッピングする

概要

クライアントによっては、ブラウザー同様に表現のメディアタイプや言語のネゴシエーションに **Accept** および **Accept-Language** ヘッダーを使用できないものがあります。**RESTEasy** はこの問題に対処するため、ファイル名のサフィックスをメディアタイプや言語にマッピングすることができます。次の手順に従って、**web.xml** ファイルにて言語をファイル拡張子にマッピングします。

手順12.5 web.xml ファイルにてファイル拡張子を言語にマッピングする

1. テキストエディターでアプリケーションの **web.xml** ファイルを開きます。
2. コンテキストパラメーター **resteasy.language.mappings** をファイルの **web-app** タグ内に追加します。

```
<context-param>
    <param-name>resteasy.language.mappings</param-name>
</context-param>
```

3. パラメーター値を設定します。マッピングはコンマ区切りリストを形成します。各マッピングは：で区切られます。

例12.14 マッピングの例

```
<context-param>
    <param-name>resteasy.language.mappings</param-name>
```

```
    <param-value> en : en-US, es : es, fr : fr</param-name>
</context-param>
```

バグを報告する

12.11.3. RESTEasy 対応メディアの種類

表12.5 メディアの種類

メディアの種類	Java 型
application/*+xml, text/*+xml, application/*+json, application/*+fastinfoset, application/atom+*	JaxB アノテーション付きクラス
application/*+xml, text/*+xml	org.w3c.dom.Document
/	java.lang.String
/	java.io.InputStream
テキスト／プレーン	プリミティブ、java.lang.String、ストリングコンストラクターを持つ型、インプット向けの静的 valueOf(String) メソッド、アウトプット向けの toString()
/	javax.activation.DataSource
/	byte[]
/	java.io.File
application/x-www-form-urlencoded	javax.ws.rs.core.MultivaluedMap

バグを報告する

12.12. RESTEASY JAVASCRIPT API

12.12.1. RESTEasy JavaScript API について

RESTEasy は AJAX 呼び出しを使用する JavaScript を生成して JAX-RS 操作を呼び出します。各 JAX-RS リソースクラスは、宣言するクラスまたはインターフェースを同じ名前の JavaScript オブジェクトを生成します。JavaScript オブジェクトには各 JAX-RS メソッドがプロパティーとして含まれます。

例12.15 単純な JAX-RS JavaScript API の例

```
@Path("/")
public interface X{
```

```
 @GET
 public String Y();
 @PUT
 public void Z(String entity);
}
```

上記のインターフェースは JavaScript API のプロパティーになる メソッド Y と Z を下記のように定義します。

```
var X = {
 Y : function(params){â ¦},
 Z : function(params){â ¦}
};
```

各 JavaScript API メソッドは、任意のオブジェクトを単一のパラメーターとして取ります。このパラメーターでは、各プロパティーは名前または API パラメータープロパティーによって識別される通り クッキー、ヘッダー、パス、クエリ、形式パラメーターのいずれかになります。プロパティーは「RESTEasy Javascript API パラメーター」 にあります。

バグを報告する

12.12.2. RESTEasy JavaScript API サーブレットの有効化

概要

RESTEasy JavaScript API はデフォルトでは有効になっていません。次の手順に従い、**web.xml** ファイルを使用して有効にします。

手順12.6 web.xml を編集して RESTEasy JavaScript API を有効にする

1. テキストエディターでアプリケーションの **web.xml** ファイルを開きます。
2. 以下の設定をファイルの **web-app** タグ内に追加します。

```
<servlet><servlet-name>RESTEasy JSAPI</servlet-name><servlet-
class>org.jboss.resteasy.jsapi.JSAPIServlet</servlet-class>
</servlet><servlet-mapping><servlet-name>RESTEasy JSAPI</servlet-
name><url-pattern>/URL</url-pattern></servlet-mapping>
```

バグを報告する

12.12.3. RESTEasy Javascript API パラメーター

表12.6 パラメータープロパティー

プロパティ	デフォルト値	説明
$entity		PUT、POST リクエストとして送信するエンティティー。
$contentType		Content-Type ヘッダーとして送信されるボディーエンティティーの MIME タイプ。@Consumes アノテーションによって判断されます。
$accepts	*/*	Accept ヘッダーとして送信される許可された MIME タイプ。@Provides アノテーションによって判断されます。
$callback		非同期呼び出しの関数 (httpCode、xmlHttpRequest、value) に設定されます。指定がない場合、呼び出しは同期となり、値が返されます。
@apiURL		最後のスラッシュを含まない JAX-RS エンドポイントのベース URI に設定されます。
$username		ユーザー名とパスワードが設定されている場合、設定されているユーザー名とパスワードがリクエストの認証情報に使用されます。
$password		ユーザー名とパスワードが設定されている場合、設定されているユーザー名とパスワードがリクエストの認証情報に使用されます。

バグを報告する

12.12.4. JavaScript API を用いた AJAX クエリの構築

概要

RESTEasy JavaScript API を手作業で使用してリクエストを構築することができます。このトピックではこの動作の例について取り上げます。

> **例12.16 REST オブジェクト**
>
> REST オブジェクトを使用して RESTEasy JavaScript API クライアントの動作をオーバーライドできます。
>
> ```
> // Change the base URL used by the API:
> REST.apiURL = "http://api.service.com";
> ```

```
// log everything in a div element
REST.log = function(text){
 jQuery("#log-div").append(text);
};
```

REST オブジェクトには次の読み書きプロパティーが含まれています。

apiURL

デフォルトで JAX-RS ルート URL に設定されます。リクエストを構築する時にすべての JavaScript クライアント API 関数によって使用されます。

log

RESTEasy クライアント API のログを受信するため function(string) に設定されます。クライアント API をデバッグし、見える場所にログを置きたい場合に便利です。

例12.17 REST.Request クラス

REST.Request クラスを使用してカスタムリクエストを構築することができます。

```
var r = new REST.Request();
r.setURI("http://api.service.com/orders/23/json");
r.setMethod("PUT");
r.setContentType("application/json");
r.setEntity({id: "23"});
r.addMatrixParameter("JSESSIONID", "12309812378123");
r.execute(function(status, request, entity){
 log("Response is "+status);
});
```

バグを報告する

12.12.5. REST.Request クラスメンバー

表12.7 REST.Request クラス

メンバー	説明
execute(callback)	現在のオブジェクトに設定されたすべての情報を持つリクエストを実行します。値は任意の引数コールバックへ渡され、返されません。
setAccepts(acceptHeader)	Accept リクエストヘッダーを設定します。デフォルトは */* です。
setCredentials(username, password)	リクエストの認証情報を設定します。

メンバー	説明
setEntity(entity)	リクエストエンティティーを設定します。
setContentType(contentTypeHeader)	Content-Type リクエストヘッダーを設定します。
setURI(uri)	リクエスト URI を設定します。絶対 URI でなければなりません。
setMethod(method)	リクエストメソッドを設定します。デフォルトは GET です。
setAsync(async)	リクエストが非同期であるべきかどうかを制御します。デフォルトは true です。
addCookie(name, value)	リクエストを実行する時に現在のドキュメントに特定のクッキーを設定します。これはブラウザーで永続化されます。
addQueryParameter(name, value)	クエリパラメーターを URI のクエリ部分に追加します。
addMatrixParameter(name, value)	リクエスト URI の最後のパスセグメントへマトリックスパラメーター (パスパラメーター) を追加します。
addHeader(name, value)	リクエストヘッダーを追加します。

バグを報告する

12.13. RESTEASY 非同期ジョブサービス

12.13.1. RESTEasy 非同期ジョブサービスについて

RESTEasy 非同期ジョブサービスは HTTP プロトコルに非同期の動作を追加するものです。HTTP は同期的なプロトコルですが、非同期呼び出しについてわずかな知識を持っています。HTTP 1.1 の応答コード 202 では、「許可」はサーバーが処理の応答を受信し許可したことを意味しますが、処理は完了していません。非同期ジョブサービスはここにビルドされます。

このサービスを有効にするには、「非同期ジョブサービスの有効化」を参照してください。サービスの例については 「RESTEasy 向けに非同期ジョブを設定」を参照してください。

バグを報告する

12.13.2. 非同期ジョブサービスの有効化

手順12.7 web.xml ファイルの変更

- `web.xml` ファイルで非同期ジョブサービスを有効化
  -

```
<context-param>
    <param-name>resteasy.async.job.service.enabled</param-name>
    <param-value>true</param-value>
</context-param>
```

結果

非同期ジョブサービスは有効化されました。設定オプションについては、「非同期ジョブサービスの設定パラメーター」 を参照してください。

バグを報告する

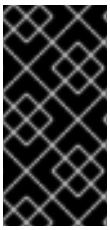
12.13.3. RESTEasy 向けに非同期ジョブを設定

概要

このトピックでは、RESTEasy を使った非同期ジョブのクエリパラメーター例について見ていきます。

警告

ポータブルに実装ができないため、ロールベースのセキュリティは非同期ジョブサービスでは機能しません。非同期ジョブサービスを利用する場合、アプリケーションセキュリティは `web.xml` ファイルで XML 宣言しなければなりません。



重要

GET、DELETE、PUT メソッドを非同期的に呼び出すことができますが、これらのメソッドの HTTP 1.1 コントラクトに違反することになります。これらの呼び出しは複数回呼び出された場合にリソースの状態を変更しないこともありますが、呼び出しごとに新しいジョブエントリとしてサーバーの状態を変更します。

例12.18 非同期パラメーター

asynch クエリパラメーターを使いバックグラウンドで呼び出しを実行します。バックグラウンドメソッドのレスポンスの場所を参照する URL が含まれるロケーションヘッダーとあわせ、202 Accepted レスポンスが返されます。

```
POST http://example.com/myservice?asynch=true
```

上記の例では、202 Accepted レスポンスと、ロケーションヘッダー (バックグラウンドメソッドのレスポンスの場所を参照する URL が含まれる) を返します。ロケーションヘッダーのサンプルを以下に示しています。

```
HTTP/1.1 202 Accepted
Location: http://example.com/asynch/jobs/3332334
```

URI は以下の形式を取ります。

```
/asynch/jobs/{job-id}?wait={millisconds}|nowait=true
```

GET、POST、DELETE 操作をこの URL で実行可能です。

- ジョブが完了した場合、GET はレスポンスとして呼び出された JAX-RS リソースメソッドを返します。ジョブが完了していない場合、この GET が 202 Accepted レスポンスコードを返します。GET を呼び出してもジョブは削除されないので、複数回呼び出すことができます。
- POST はジョブのレスポンスを読み取り、完了した場合はジョブを削除します。
- DLETE はジョブのキューを手動消去するために呼び出されます。

注記

ジョブのキューがいっぱいの場合は、DELETE を呼び出さずに、自動的にメモリから最初のジョブをエビクトします。

例12.19 Wait / Nowait

GET および POST 操作では、**wait** や**nowait** を使うことで最大待機時間を定義できます。**wait** パラメーターを指定されていない場合、この操作はデフォルトの**nowait=true** となり、ジョブが完了していない場合も待機しません。**wait** パラメーターをミリ秒で定義します。

```
POST http://example.com/asynch/jobs/122?wait=3000
```

例12.20 Oneway パラメーター

RESTEasy は **oneway** クエリパラメーターを使うことで、fire/forget ジョブに対応しています。

```
POST http://example.com/myservice?oneway=true
```

上記の例は、202 Accepted のレスポンスを返しますが、ジョブは作成されません。

バグを報告する

12.13.4. 非同期ジョブサービスの設定パラメーター

概要

下表は非同期ジョブサービスの設定可能なコンテキストパラメーターと詳細を表しています。これらのパラメーターは **web.xml** ファイルに設定することができます。

表12.8 設定パラメーター

パラメーター	説明

パラメーター	説明
resteasy.async.job.service.max.job.results	一度にメモリーに保持できるジョブ結果の数です。デフォルト値は 100 になります。
resteasy.async.job.service.max.wait	クライアントがジョブをクエリする時のジョブの最大待機時間です。デフォルト値は 300000 です。
resteasy.async.job.service.thread.pool.size	ジョブを実行するバックグラウンドスレッドのスレッドプールサイズです。デフォルト値は 100 になります。
resteasy.async.job.service.base.path	ジョブの URI のベースパスを設定します。デフォルト値は /asynch/jobs になります。

例12.21 非同期ジョブ設定の例

```
<web-app>
    <context-param>
        <param-name>resteasy.async.job.service.enabled</param-name>
        <param-value>true</param-value>
    </context-param>

    <context-param>
        <param-name>resteasy.async.job.service.max.job.results</param-
name>
        <param-value>100</param-value>
    </context-param>
    <context-param>
        <param-name>resteasy.async.job.service.max.wait</param-name>
        <param-value>300000</param-value>
    </context-param>
    <context-param>
        <param-name>resteasy.async.job.service.thread.pool.size</param-
name>
        <param-value>100</param-value>
    </context-param>
    <context-param>
        <param-name>resteasy.async.job.service.base.path</param-name>
        <param-value>/asynch/jobs</param-value>
    </context-param>

    <listener>
        <listener-class>
            org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
        </listener-class>
    </listener>

    <servlet>
        <servlet-name>Resteasy</servlet-name>
        <servlet-class>

org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
```

```
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Resteasy</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

バグを報告する

12.14. RESTEASY JAXB

12.14.1. JAXB デコレーターの作成

概要

RESTEasy の JAXB プロバイダーはマーシャラーおよびアンマーシャラーインスタンスを修飾するプラグ可能な方法を提供します。作成されたアノテーションによってマーシャラーまたはアンマーシャラーインスタンスが発生します。本トピックでは RESTEasy を用いて JAXB デコレーターを作成する手順を取り上げます。

手順12.8 RESTEasy による JAXB デコレーターの作成

1. **プロセッサークラスの作成**

    a. DecoratorProcessor<Target, Annotation> を実装するクラスを作成します。ターゲットは JAXB マーシャラーまたはアンマーシャラーのクラスになります。アノテーションは手順 2 で作成されます。

    b. @DecorateTypes アノテーションをクラスに付け、デコレーターが修飾する必要がある MIME タイプ を宣言します。

    c. **`decorate`** 関数内でプロパティーまたは値を設定します。

例12.22 プロセッサークラスの例

```
import org.jboss.resteasy.core.interception.DecoratorProcessor;
import org.jboss.resteasy.annotations.DecorateTypes;

import javax.xml.bind.Marshaller;
import javax.xml.bind.PropertyException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.Produces;
import java.lang.annotation.Annotation;

@DecorateTypes({"text/*+xml", "application/*+xml"})
public class PrettyProcessor implements
DecoratorProcessor<Marshaller, Pretty>
{
    public Marshaller decorate(Marshaller target, Pretty
annotation,
```

```
    Class type, Annotation[] annotations, MediaType mediaType)
     {
 target.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
Boolean.TRUE);
     }
}
```

2. アノテーションの作成

   a. @Decorator アノテーションが付けられたカスタムインターフェースを作成します。

   b. @Decorator アノテーションのプロセッサーとターゲットを宣言します。プロセッサーは手順 1 で作成されています。ターゲットは JAXB マーシャラーまたはアンマーシャラーのクラスになります。

例12.23 アノテーションの例

```
import org.jboss.resteasy.annotations.Decorator;

@Target({ElementType.TYPE, ElementType.METHOD,
ElementType.PARAMETER, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Decorator(processor = PrettyProcessor.class, target =
Marshaller.class)
public @interface Pretty {}
```

3. 手順 2 で作成されたアノテーションを関数に追加し、マーシャルされた時に入力か出力が修飾されるようにします。

結果

JAXB デコレーターが作成され、JAX-RS Web サービス内で適用されます。

バグを報告する

12.15. RESTEASY ATOM サポート

12.15.1. Atom API とプロバイダーについて

RESTEasy Atom API とプロバイダーは Atom を表すためRESTEasy が定義する簡単なオブジェクトモデルです。API の主なクラスは **org.jboss.resteasy.plugins.providers.atom** パッケージにあります。RESTEasy は JAXB を使用してAPI をマーシャルおよびアンマーシャルします。 プロバイダーは JAXB ベースで、XML を使用した atom オブジェクトの送信のみに限定されません。RESTEasy が持つ全 JAXB プロバイダーは、JSON や fastinfoset などの Atom API とプロバイダーによる再使用が可能です。API の詳細は javadocs を参照してください。

バグを報告する

第13章 JAX-WS WEB サービス

13.1. JAX-WS WEB サービスについて

Java API for XML Web Services (JAX-WS) は Java Enterprise Edition (EE) プラットフォームに含まれる API で、Web サービスの作成に使用されます。Web サービスとは、主に XML や他の構造化されたテキスト形式での情報交換など、ネットワーク上で通信を行うためのアプリケーションのことです。Web サービスはプラットフォームから独立しています。通常の JAX-WS アプリケーションはクライアント／サーバーモデルを使用します。サーバーコンポーネントは *Web サービスエンドポイント*と呼ばれます。

JAX-WS には JAX-RS と呼ばれるプロトコルを使用する小型で単純な Web サービス向けものがあります。JAX-RS は *Representational State Transfer*(REST) のプロトコルです。JAX-RS アプリケーションは通常軽量で HTTP プロトコルのみに依存して通信を行います。**WS-Notification**、**WS-Addressing**、**WS-Policy**、**WS-Security**、**WS-Trust** などの Web サービス指向プロトコルは JAX-WS によってサポートが容易になります。これらのプロトコルはメッセージアーキテクチャーやメッセージ形式を定義する *シンプルオブジェクトアクセスプロトコル (SOAP)*と呼ばれる特殊な XML 言語を使用して通信します。

JAX-WS Web サービスには提供する操作の機械可読な記述も含まれます。これは特殊な XML 文書型である *Web サービス記述言語 (WSDL)*で書かれています。

Web サービスエンドポイントは **WebService** インターフェースと **WebMethod** インターフェースを実装するクラスによって構成されます。

Web サービスクライアントは、WSDL 定義によって生成される*スタブ*と呼ばれる複数のクラスに依存するクライアントによって構成されます。

JAX-WS Web サービスでは、正式なコントラクトが確立され、Web サービスが提供するインターフェースが記述されます。通常、コントラクトは WSDL で書かれますが、SOAP メッセージで書くことも可能です。通常、Web サービスのアーキテクチャーはトランザクション、セキュリティー、メッセージング、コーディネーションなどのビジネス要件に対応します。JBoss Enterprise Application Platform はこれらビジネスの懸念を処理するメカニズムを提供します。

*Web サービス記述言語 (WSDL)*は Web サービスや Web サービスへのアクセス方法を記述するために使用される XML ベースの言語です。Web サービス自体は Java や他のプログラミング言語で書かれます。WSDL 定義はインターフェースへの参照、ポート定義、ネットワーク上で他の Web サービスが対話する方法の指示によって構成されます。Web サービスは*シンプルオブジェクトアクセスプロトコル (SOAP)*を用いて通信します。このタイプの Web サービスは*Representative State Transfer (REST)*の設計原理を用いて構築された *RESTful Web サービス*とは対照的です。RESTful Web サービスでは WSDL や SOAP を使用する必要はありませんが、他のサービスと対話する方法は HTTP プロトコルの構造に依存して定義されます。

JBoss Enterprise Application Platform には JAX-WS Web サービスエンドポイントのデプロイメントに対するサポートが含まれています。このサポートは JBossWS によって提供されます。エンドポイントの設定やハンドラーチェーン、ハンドラーなどの Web サービスサブシステムの設定は **webservices** サブシステムより提供されます。

作業例

JBoss Enterprise Application Platform 6 のクイックスタートには完全に機能する JAX-WS Web サービスアプリケーションが複数含まれています。これらの例には以下が含まれています。

- wsat-simple
- wsba-coordinator-completion-simple

- wsba-participant-completion-simple

バグを報告する

13.2. スタンドアロンWEBSERVICES サブシステムの設定

JBoss Enterprise Application Platform 6 にデプロイされた Web サービスの振る舞いを制御する **webservices** サブシステムには、数多くの設定オプションを使用することができます。管理 CLI スクリプト (***EAP_HOME*/bin/jboss-cli.sh** または ***EAP_HOME*/bin/jboss-cli.bat**) 内の各要素を変更するためのコマンドが提供されています。スタンドアロンサーバーには、**/profile=default** の部分は削除します。また、管理対象ドメイン上の異なるプロファイルには、この部分を修正してサブシステムを変更してください。

エンドポイントアドレスの公開

エンドポイントで公開している WSDL コントラクトの **<soap:address>** 要素を書き換えることができます。この機能は、各エンドポイントのクライアントに対してアドバタイズされたサーバーアドレスを制御するのに使用することができます。以下のオプション要素はそれぞれ、必要に応じて修正することができます。これらのいずれかの要素を修正した場合には、サーバーを再起動する必要があります。

表13.1 公開されるエンドポイントアドレスの設定要素

要素	説明	CLI コマンド
modify-wsdl-address	WSDL アドレスを常に変更するかどうかを設定します。true に指定した場合には、**<soap:address>** の内容は常に上書きされます。false に指定した場合には、**<soap:address>** の内容は URL が有効でない場合のみ上書きされます。使用する値は、**wsdl-host**、**wsdl-port**、および**wsdl-secure-port** で、以下に説明を記載しています。	**/profile=default/subsystem=webservices/:write-attribute(name=modify-wsdl-address,value=true)**
wsdl-host	**<soap:address>** を書き換える際に使用するホスト名/IP アドレス。**wsdl-host** を文字列 **jbossws.undefined.host** に設定すると、**<soap:address>** 書き換えの際にリクエスターのホストが使用されます。	**/profile=default/subsystem=webservices/:write-attribute(name=wsdl-host,value=10.1.1.1)**
wsdl-port	SOAP アドレスの書き換えに使用される HTTP ポートを明示的に定義する整数。未定義の場合には、インストール済みの HTTP コネクターの一覧に対してクエリを実行することによってHTTP ポートが識別されます。	**/profile=default/subsystem=webservices/:write-attribute(name=wsdl-port,value=8080)**

要素	説明	CLI コマンド
wsdl-secure-port	SOAP アドレスの書き換えに使用される HTTPS ポートを明示的に定義する整数。未定義の場合には、インストール済みの HTTPS コネクターの一覧に対してクエリを実行することによって HTTPS ポートが識別されます。	**/profile=default/subsystem=webservices/:write-attribute(name=wsdl-secure-port,value=8443)**

事前定義済みのエンドポイント設定

エンドポイントの実装が参照可能なエンドポイント設定を定義することができます。その用途の一つとして、**@org.jboss.ws.api.annotation.EndpointConfig** のアノテーションが付いた所定のエンドポイント設定でマークされた任意の WS エンドポイントに所定のハンドラーを追加することができます。

JBoss Enterprise Application Platform にはデフォルトの **Standard-Endpoint-Config** が含まれています。また、カスタム設定の例である **Recording-Endpoint-Config** も含まれています。これは、レコーディングハンドラーの例を提供します。**Standard-Endpoint-Config** は、どの設定とも関連付けされていないエンドポイントに自動的に使用されます。

管理 CLI を使用して **Standard-Endpoint-Config** を読み取るには、次のコマンドを実行します。

```
/profile=default/subsystem=webservices/endpoint-config=Standard-Endpoint-Config/:read-resource(recursive=true,proxies=false,include-runtime=false,include-defaults=true)
```

エンドポイントの設定

エンドポイントの設定は、管理 API では **endpoint-config** と呼ばれており、**post-handler-chain**、**post-handler-chain** および特定のエンドポイントに適用される一部のプロパティーが含まれます。endpoint config の読み取りには以下のコマンドを使用します。

例13.1 endpoint config の読み取り

```
/profile=default/subsystem=webservices/endpoint-config=Recording-Endpoint-Config:read-resource
```

例13.2 endpoint config の追加

```
/profile=default/subsystem=webservices/endpoint-config=My-Endpoint-Config:add
```

ハンドラーチェーン

各 endpoint config は **PRE** および **POST** ハンドラーチェーンと関連付けすることができます。各ハンドラーチェーンには、JAXWS に準拠したハンドラーを追加することが可能です。送信メッセージの場合は、**@HandlerChain** アノテーションなどの 標準的な JAXWS の方法を使用してエンドポイントに接続されるハンドラーよりも前に、PRE ハンドラーチェーンのハンドラーが実行されます。POST ハンド

ラーチェーンのハンドラーは、通常のエンドポイントハンドラーの後に実行されます。 受信メッセージの場合は、その逆が適用されます。JAX-WS は、XML ベース Web サービス向けの標準 API で、http://jcp.org/en/jsr/detail?id=224 に文書化されています。

ハンドラーチェーンには、チェーンの開始をトリガーするプロトコルを設定する **protocol-binding** 属性を追加することも可能です。

例13.3 ハンドラーチェーンの読み取り

```
/profile=default/subsystem=webservices/endpoint-config=Recording-
Endpoint-Config/pre-handler-chain=recording-handlers:read-resource
```

例13.4 ハンドラーチェーンの追加

```
/profile=default/subsystem=webservices/endpoint-config=My-Endpoint-
Config/post-handler-chain=my-handlers:add(protocol-
bindings="##SOAP11_HTTP")
```

ハンドラー

JAXWS ハンドラーは、ハンドラーチェーン内の子エレメント **<handler>** です。このハンドラーは、ハンドラークラスの完全修飾クラス名である **class** 属性を取ります。エンドポイントがデプロイされる際には、参照するデプロイメントごとにそのクラスのインスタンスが作成されます。デプロイメントクラスローダーまたは **org.jboss.as.webservices.server.integration** モジュール用のクラスローダーのいずれかがハンドラークラスをロード可能である必要があります。

例13.5 ハンドラーの読み取り

```
/profile=default/subsystem=webservices/endpoint-config=Recording-
Endpoint-Config/pre-handler-chain=recording-
handlers/handler=RecordingHandler:read-resource
```

例13.6 ハンドラーの追加

```
/profile=default/subsystem=webservices/endpoint-config=My-Endpoint-
Config/post-handler-chain=my-handlers/handler=foo-
handler:add(class="org.jboss.ws.common.invocation.RecordingServerHandler
")
```

Web サービスについてのランタイム情報

Web コンテキストや WSDL URL などの Web サービスのランタイム情報は、エンドポイント自体を問い合わせることによって表示することができます。 * の文字を使用すると、全エンドポイントを一度に問い合わせることができます。以下の 2 つの情報は、管理対象ドメインのサーバーとスタンドアロンサーバーに対するコマンドを示しています。

例13.7 管理対象ドメイン内のサーバーでの全エンドポイントに関するランタイム情報の表示

このコマンドは、管理対象ドメイン内の物理ホスト **master** でホストされた **server-one** という名前のサーバーですべてのエンドポイントに関する情報を表示します。

```
/host=master/server=server-
one/deployment="*"/subsystem=webservices/endpoint="*":read-resource
```

例13.8 スタンドアロンサーバーでの全エンドポイントに関するランタイム情報の表示

このコマンドは、**master** という名前の物理ホストの **server-one** という名前のスタンドアロンサーバーですべてのエンドポイントに関する情報を表示します。

```
/host=master/server=server-
one/deployment="*"/subsystem=webservices/endpoint="*":read-resource
```

例13.9 エンドポイント情報の例

この出力の仮説例は以下のとおりです。

```
{
   "outcome" => "success",
   "result" => [{
       "address" => [
           ("deployment" => "jaxws-samples-handlerchain.war"),
           ("subsystem" => "webservices"),
           ("endpoint" => "jaxws-samples-handlerchain:TestService")
       ],
       "outcome" => "success",
       "result" => {
           "class" =>
"org.jboss.test.ws.jaxws.samples.handlerchain.EndpointImpl",
           "context" => "jaxws-samples-handlerchain",
           "name" => "TestService",
           "type" => "JAXWS_JSE",
           "wsdl-url" => "http://localhost:8080/jaxws-samples-
handlerchain?wsdl"
       }
   }]
}
```

バグを報告する

13.3. JAX-WS WEB サービスエンドポイント

13.3.1. JAX-WS Web サービスエンドポイントについて

ここでは、JAX-WS Web サービスエンドポイントおよびそれに付随する概念について取り上げます。JAX-WS Web サービスエンドポイントは、Web サービスのサーバーコンポーネントです。クライアントおよびその他の Web サービスは *Simple Object Access Protocol (SOAP)* と呼ばれる XML 言語を使用

し、HTTP プロトコルを介して通信します。 エンドポイント自体は JBoss Enterprise Application Platform コンテナにデプロイされます。

WSDL 記述子は手動で作成することが可能です。また、JAX-WS アノテーションを使用して自動的に作成することもできます。これは、より標準的な使用パターンです。

エンドポイント実装 Bean には JAX-WS アノテーションが付けられ、サーバーにデプロイされます。サーバーは、抽象コントラクトをクライアントが使用できるように WSDL 形式で生成し公開します。マーシャリングおよびアンマーシャリングはすべて *Java Architecture for XML Binding (JAXB)*サービスへ委譲されます。

エンドポイント自体は POJO (Plain Old Java Object) または Java EE Web アプリケーションである場合があります。また、EJB3 ステートレスセッション Bean を使用してエンドポイントを公開することもできます。これは Web アーカイブ (WAR) ファイルにパッケージ化されます。*Java Service Endpoint (JSE)* と呼ばれるエンドポイントのパッケージングの仕様は、http://jcp.org/aboutJava/communityprocess/pfd/jsr181/index.html に記載の JSR-181 で定義されています。

開発要件

Web サービスは、http://www.jcp.org/en/jsr/summary?id=181 に記載の JAX-WS API および Web サービスメタデータ仕様要件を満たしている必要があります。有効な実装は以下の要件を満たします。

- **`javax.jws.WebService`** アノテーションが含まれます。
- メソッドのパラメーターおよび戻り値の型はすべて JAXB 2.0 の仕様 JSR-222 との互換性があります。詳しくは http://www.jcp.org/en/jsr/summary?id=222 を参照してください。

例13.10 POJO エンドポイントの例

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
{
   @WebMethod
   public String echo(String input)
   {
      ...
   }
}
```

例13.11 Web サービスエンドポイントの例

```
<web-app ...>
  <servlet>
    <servlet-name>TestService</servlet-name>
    <servlet-
class>org.jboss.test.ws.jaxws.samples.jsr181pojo.JSEBean01</servlet-
class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestService</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

例13.12 EJB 内のエンドポイントの公開

この EJB3 ステートレスセッション Bean は、リモートインターフェース上に同じメソッドをエンドポイントの操作として公開します。

```
@Stateless
@Remote(EJB3RemoteInterface.class)
@RemoteBinding(jndiBinding = "/ejb3/EJB3EndpointInterface")

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class EJB3Bean01 implements EJB3RemoteInterface
{
   @WebMethod
   public String echo(String input)
   {
      ...
   }
}
```

エンドポイントプロバイダー

JAX-WS サービスは通常、Java サービスエンドポイントインターフェース (SEI) を実装します。これは、WSDL ポートタイプから直接もしくはアノテーションを使用してマッピングすることが可能です。この SEI は、Java オブジェクトとそれらの XML 表現の間の詳細情報を隠すハイレベルの抽象化を提供します。ただし、サービスが XML メッセージレベルで稼働する機能を必要とする場合があります。エンドポイント **Provider** インターフェースはこの機能を Web サービスに提供し、その Web サービスが機能を実装します。

エンドポイントの使用とアクセス

Web サービスをデプロイした後、WSDL を使用して、アプリケーションの基盤となるコンポーネントスタブを作成することが可能です。これでアプリケーションはエンドポイントにアクセスして作業を行うことができます。

作業例

JBoss Enterprise Application Platform 6 のクイックスタートには完全に機能する JAX-WS Web サービスアプリケーションが複数含まれています。これらの例には以下が含まれています。

- wsat-simple
- wsba-coordinator-completion-simple
- wsba-participant-completion-simple

バグを報告する

13.3.2. JAX-WS Web サービスエンドポイントの書き込みとデプロイ

はじめに

本トピックでは、シンプルな JAX-WS サービスエンドポイントの開発について説明します。JAX-WS サービスエンドポイントは、JAX-WS クライアントからの要求に応答し、WSDL 定義を自らに大して公開するサーバー側のコンポーネントです。JAX-WS サービスエンドポイントに関する詳細については、「JAX-WS の共通 API リファレンス」および JBoss Enterprise Application Platform 6 に同梱されている Javadoc 形式の API ドキュメントバンドルを参照してください。

開発要件

Web サービスは、http://www.jcp.org/en/jsr/summary?id=181 に記載の JAX-WS API および Web サービスメタデータの仕様要件を満たしている必要があります。有効な実装は以下の要件を満たします:

- **`javax.jws.WebService`** アノテーションが含まれます。
- メソッドのパラメーターおよび戻り値の型はすべて JAXB 2.0 の仕様 JSR-222 との互換性があります。詳しくは http://www.jcp.org/en/jsr/summary?id=222 を参照してください。

例13.13 サービス実装の例

```
package org.jboss.test.ws.jaxws.samples.retail.profile;

import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;

@Stateless
@WebService(
   name="ProfileMgmt",
   targetNamespace = "http://org.jboss.ws/samples/retail/profile",
   serviceName = "ProfileMgmtService")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
public class ProfileMgmtBean {

   @WebMethod
   public DiscountResponse getCustomerDiscount(DiscountRequest request)
{
      return new DiscountResponse(request.getCustomer(), 10.00);
   }
}
```

例13.14 XML ペイロードの例

上記の例に記載の **`ProfileMgmtBean`** Bean によって使用される **`DiscountRequest`** クラスの例は以下のとおりです。アノテーションは詳細のために含まれています。通常、JAXB のデフォルト設定は妥当なので指定する必要はありません。

```
package org.jboss.test.ws.jaxws.samples.retail.profile;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;
```

```
import org.jboss.test.ws.jaxws.samples.retail.Customer;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(                                                    (1)
  name = "discountRequest",
  namespace="http://org.jboss.ws/samples/retail/profile",
  propOrder = { "customer" }
)
public class DiscountRequest {

   protected Customer customer;

   public DiscountRequest() {
   }

   public DiscountRequest(Customer customer) {
      this.customer = customer;
   }

   public Customer getCustomer() {
      return customer;
   }

   public void setCustomer(Customer value) {
      this.customer = value;
   }

}
```

より複雑なマッピングが可能です。詳しい情報は http://java.sun.com/webservices/jaxb/ に記載の JAXB API 仕様を参照してください。

デプロイメントのパッケージ

実装クラスは **JAR** デプロイメントにラッピングされます。実装クラスおよびサービスエンドポイントインターフェースに対するアノテーションからデプロイメントに必要な任意のメタデータが取得されます。管理 CLI または管理インターフェースを使用して JAR をデプロイすると、HTTP エンドポイントが自動的に作成されます。

以下の一覧は、EJB Web サービスの JAR デプロイメントの適正な構造の例を示しています。

例13.15 Web サービスデプロイメントの JAR 構造の例

```
[user@host ~]$ jar -tf jaxws-samples-retail.jar
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.class
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.class
org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtBean.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.class
org/jboss/test/ws/jaxws/samples/retail/profile/package-info.class
```

バグを報告する

13.4. JAX-WS WEB サービスクライアント

13.4.1. JAX-WS Web サービスの使用とアクセス

手動または JAX-WS アノテーションを使用して Web サービスエンドポイントを作成した後には、WSDL にアクセスして、Web サービスと通信を行う基本的なクライアントアプリケーションを作成することができます。公開されている WSDL からの Java コード生成プロセスは、Web サービスの消費と呼ばれます。これ は 2 段階で実行されます。

1. クライアントアーティファクトの作成
2. サービススタブの構築
3. エンドポイントへのアクセス

クライアントアーティファクトの作成

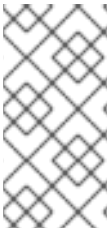
クライアントアーティファクトを作成する前に、WSDL コントラクトを作成しておく必要があります。以下の WSDL コントラクトは、以降、本トピックで例として使用します。

例13.16 WSDL コントラクトの例

```
<definitions
    name='ProfileMgmtService'
    targetNamespace='http://org.jboss.ws/samples/retail/profile'
    xmlns='http://schemas.xmlsoap.org/wsdl/'
    xmlns:ns1='http://org.jboss.ws/samples/retail'
    xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
    xmlns:tns='http://org.jboss.ws/samples/retail/profile'
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

   <types>

      <xs:schema targetNamespace='http://org.jboss.ws/samples/retail'
                 version='1.0'
xmlns:xs='http://www.w3.org/2001/XMLSchema'>
         <xs:complexType name='customer'>
            <xs:sequence>
               <xs:element minOccurs='0' name='creditCardDetails'
type='xs:string'/>
               <xs:element minOccurs='0' name='firstName'
type='xs:string'/>
               <xs:element minOccurs='0' name='lastName'
type='xs:string'/>
            </xs:sequence>
         </xs:complexType>
      </xs:schema>

      <xs:schema
          targetNamespace='http://org.jboss.ws/samples/retail/profile'
          version='1.0'
          xmlns:ns1='http://org.jboss.ws/samples/retail'
          xmlns:tns='http://org.jboss.ws/samples/retail/profile'
```

```
          xmlns:xs='http://www.w3.org/2001/XMLSchema'>

         <xs:import namespace='http://org.jboss.ws/samples/retail'/>
         <xs:element name='getCustomerDiscount'
                     nillable='true' type='tns:discountRequest'/>
         <xs:element name='getCustomerDiscountResponse'
                     nillable='true' type='tns:discountResponse'/>
         <xs:complexType name='discountRequest'>
            <xs:sequence>
               <xs:element minOccurs='0' name='customer'
type='ns1:customer'/>

            </xs:sequence>
         </xs:complexType>
         <xs:complexType name='discountResponse'>
            <xs:sequence>
               <xs:element minOccurs='0' name='customer'
type='ns1:customer'/>
               <xs:element name='discount' type='xs:double'/>
            </xs:sequence>
         </xs:complexType>
      </xs:schema>

   </types>

   <message name='ProfileMgmt_getCustomerDiscount'>
      <part element='tns:getCustomerDiscount'
name='getCustomerDiscount'/>
   </message>
   <message name='ProfileMgmt_getCustomerDiscountResponse'>
      <part element='tns:getCustomerDiscountResponse'
            name='getCustomerDiscountResponse'/>
   </message>
   <portType name='ProfileMgmt'>
      <operation name='getCustomerDiscount'
                 parameterOrder='getCustomerDiscount'>

         <input message='tns:ProfileMgmt_getCustomerDiscount'/>
         <output
message='tns:ProfileMgmt_getCustomerDiscountResponse'/>
      </operation>
   </portType>
   <binding name='ProfileMgmtBinding' type='tns:ProfileMgmt'>
      <soap:binding style='document'
                    transport='http://schemas.xmlsoap.org/soap/http'/>
      <operation name='getCustomerDiscount'>
         <soap:operation soapAction=''/>
         <input>

            <soap:body use='literal'/>
         </input>
         <output>
            <soap:body use='literal'/>
         </output>
      </operation>
   </binding>
```

```
   <service name='ProfileMgmtService'>
      <port binding='tns:ProfileMgmtBinding' name='ProfileMgmtPort'>

         <soap:address
             location='SERVER:PORT/jaxws-samples-
retail/ProfileMgmtBean'/>
      </port>
   </service>
</definitions>
```



注記

JAX-WS アノテーションを使用して Web サービスエンドポイントを作成した場合には、WSDL コントラクトは自動的に生成されるので、その URL のみが必要となります。この URL は、エンドポイントがデプロイされた後に、Web ベース管理コンソールの **Runtime** セクションの **Web** セクションから取得することができます。

wsconsume.sh または **wsconsume.bat** ツールを使用して抽象コントラクト (WSDL) を消費し、アノテーションが付いた Java クラスとそれを定義するオプションのソースを作成します。コマンドは、JBoss Enterprise Application Platform インストールの ***EAP_HOME*/bin/** ディレクトリにあります。

例13.17 wsconsume.sh コマンドの構文

```
[user@host bin]$ ./wsconsume.sh --help
WSConsumeTask is a cmd line tool that generates portable JAX-WS
artifacts from a WSDL file.

usage: org.jboss.ws.tools.cmd.WSConsume [options] <wsdl-url>

options:
    -h, --help                  Show this help message
    -b, --binding=<file>        One or more JAX-WS or JAXB binding
files
    -k, --keep                  Keep/Generate Java source
    -c  --catalog=<file>        Oasis XML Catalog file for entity
resolution
    -p  --package=<name>        The target package for generated source
    -w  --wsdlLocation=<loc>    Value to use for
@WebService.wsdlLocation
    -o, --output=<directory>    The directory to put generated artifacts
    -s, --source=<directory>    The directory to put Java source
    -t, --target=<2.0|2.1|2.2>  The JAX-WS specification target
    -q, --quiet                 Be somewhat more quiet
    -v, --verbose               Show full exception stack traces
    -l, --load-consumer         Load the consumer and exit (debug
utility)
    -e, --extension             Enable SOAP 1.2 binding extension
    -a, --additionalHeaders     Enable processing of implicit SOAP
headers
```

```
    -n, --nocompile             Do not compile generated sources
```

以下のコマンドは、**ProfileMgmtService.wsdl** ファイルからソース **.java** ファイルを生成して出力に一覧表示します。ソースには、パッケージのディレクトリ構造が使用されます。これは、**-p** スイッチで指定します。

```
[user@host bin]$  wsconsume.sh -k -p
org.jboss.test.ws.jaxws.samples.retail.profile ProfileMgmtService.wsdl
output/org/jboss/test/ws/jaxws/samples/retail/profile/Customer.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.jav
a
output/org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.j
ava
output/org/jboss/test/ws/jaxws/samples/retail/profile/package-info.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/Customer.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.clas
s
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.cla
ss
output/org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.c
lass
output/org/jboss/test/ws/jaxws/samples/retail/profile/package-info.class
```

.java ソースファイルとコンパイル済み **.class** ファイルの両方は、コマンドを実行するディレクトリー内の **output/** ディレクトリーに生成されます。

表13.2 wsconsume.sh によって作成されたアーティファクトの説明

ファイル	説明
ProfileMgmt.java	サービスエンドポイントインターフェース。
Customer.java	カスタムデータ型。
Discount*.java	カスタムデータ型。
ObjectFactory.java	JAXB XML レジストリー。
package-info.java	JAXB パッケージアノテーション。
ProfileMgmtService.java	サービスファクトリー。

wsconsume.sh コマンドは、全カスタムデータタイプ (JAXB アノテーション付きクラス)、サービスエンドポイントインターフェース、およびサービスファクトリクラスを生成します。これらのアーティファクトは、Web サービスクライアント実装の構築に使用されます。

サービススタブの構築

Web サービスクライアントは、サービススタブを使用してリモート Web サービス呼び出しの詳細情報を抽象化します。クライアントアプリケーション側には、WS 呼び出しはその他のビジネスコンポーネントと同じように見えます。この場合、サービスエンドポイントインターフェースはビジネスインターフェースとして機能し、サービススタブとして構築する際にサービスファクトリクラスは使用されません。

> **例13.18 サービススタブの構築とエンドポイントへのアクセス**
>
> 以下の例では、まず最初に WSDL ロケーションとサービス名を使用してサービスファクトリを作成し、次に **wsconsume.sh** コマンドで作成されたサービスエンドポイントインターフェースを使用して サービススタブを構築します。最終的にこのスタブはその他のビジネスインターフェースと同じように使用することができます。
>
> エンドポイントの WSDL URL は、Web ベースの管理コンソールで確認することができます。画面の左上にある **Runtime** メニュー項目を選択し、画面左下の **Deployments** メニュー項目を選びます。**Webservices** をクリックして対象のデプロイメントを選択すると詳細が表示されます。
>
> ```
> import javax.xml.ws.Service;
> [...]
> Service service = Service.create(
> new URL("http://example.org/service?wsdl"),
> new QName("MyService")
> );
> ProfileMgmt profileMgmt = service.getPort(ProfileMgmt.class);
>
> // Use the service stub in your application
> ```

バグを報告する

13.4.2. JAX-WS クライアントアプリケーションの開発

本トピックでは JAX-WS Web Service クライアントについて概説します。クライアントは JAX-WS エンドポイントと通信し、そこから作業を要求します。 JAX-WS エンドポイントは Java Enterprise Edition 6 コンテナにデプロイされています。以下で説明するクラス、メソッド、およびその他の実装に関する詳しい情報は、「JAX-WS の共通 API リファレンス」 ならびに JBoss Enterprise Application Platform 6 に同梱されている Javadocs の該当箇所を参照してください。

サービス

概要

Service は WSDL サービスを表す抽象化です。WSDL サービスは関連ポートの集合で、それぞれには特定のプロトコルおよび特定のエンドポイントアドレスにバインドされたポート型が含まれます。

通常サービスは、既存の WSDL コントラクトから残りのコンポーネントスタブが生成される時に生成されます。WSDL コントラクトはデプロイされたエンドポイントの WSDL URL を介して利用することができます。もしくは ***EAP_HOME*/bin/** ディレクトリで **wsprovide.sh** コマンドを使用して

エンドポイントソースから作成することもできます。

このようなタイプの使用法は *静的* ユースケースと呼ばれています。この場合、コンポーネントスタブの一つとして作成された **Service** クラスのインスタンスを作成します。

サービスは **Service.create** メソッドを使用して手動で作成することも可能です。このような使用法は *動的* ユースケースと呼ばれています。

使用法

静的ユースケース

JAX-WS クライアントの *静的* ユースケースは WSDL コントラクトが既にあることを前提としています。これは、外部ツールで生成したり、 JAX-WS エンドポイントの作成時に正しい JAX-WS アノテーションを使用して生成することができます。

コンポーネントスタブを生成するには、***EAP_HOME*/bin/** に格納された **wsconsume.sh** または **wsconsume.bat** のスクリプトを使用します。スクリプトは、WSDL URL またはファイルをパラメーターとして取り、ディレクトリツリー構造の複数のファイルを生成します。 **Service** を表すソースおよびクラスのファイルはそれぞれ ***CLASSNAME*_Service.java** と ***CLASSNAME*_Service.class** と名付けられます。

生成された実装クラスには、引数なしと、 2 つの引数を使用する、 2 つのパブリックコンストラクターがあります。2 つの引数はそれぞれ WSDL ロケーション (**java.net.URL**) とサービス名 (**javax.xml.namespace.QName**) を表します。

引数なしのコンストラクターは最も頻繁に使用されます。この場合、WSDL ロケーションとサービス名は WSDL に記述された設定となります。これらは、生成されたクラスを装飾する **@WebServiceClient** アノテーションから暗黙的に設定されます。

例13.19 生成されたサービスクラスの例

```
@WebServiceClient(name="StockQuoteService",
targetNamespace="http://example.com/stocks",
wsdlLocation="http://example.com/stocks.wsdl")
public class StockQuoteService extends javax.xml.ws.Service
{
   public StockQuoteService()
   {
      super(new URL("http://example.com/stocks.wsdl"), new
QName("http://example.com/stocks", "StockQuoteService"));
   }

   public StockQuoteService(String wsdlLocation, QName serviceName)
   {
      super(wsdlLocation, serviceName);
   }

   ...
}
```

動的ユースケース

動的なケースでは、スタブは自動的には生成されず、代わりに Web サービスクライアントが **Service.create** メソッドを使用して **Service** インスタンスを作成します。以下のコードフラグメントは、このプロセスの例を示しています。

例13.20 手動でのサービス作成

```
URL wsdlLocation = new URL("http://example.org/my.wsdl");
QName serviceName = new QName("http://example.org/sample",
"MyService");
Service service = Service.create(wsdlLocation, serviceName);
```

ハンドラーリゾルバー

JAX-WS は、*ハンドラー*として知られるメッセージ処理モジュール向けの柔軟性の高いプラグインフレームワークを提供します。このようなハンドラーにより、JAX-WS ランタイムシステムの機能が拡張されます。**Service** インスタンスは、サービス、ポート、プロトコルバインディングのいずれかの単位でハンドラーのセットを設定することが可能な **getHandlerResolver** メソッドと **setHandlerResolver** メソッドのペアを介して **HandlerResolver** へのアクセスを提供します。

Service インスタンスがプロキシまたは**Dispatch** インスタンスを作成する際には、現在サービスに登録されているハンドラーリゾルバーによって必要なハンドラーチェーンが作成されます。**Service** インスタンス用に設定されたハンドラーリゾルバーがそれ以降に変更されても、以前に作成されたプロキシや **Dispatch** インスタンスには影響を及ぼしません。

エグゼキューター

Service インスタンスは **java.util.concurrent.Executor** を使用して設定することができます。**Executor** はアプリケーションが要求する任意の非同期コールバックを呼び出します。**Service** の **setExecutor** メソッドと **getExecutor** のメソッドはサービス用に設定された **Executor** を変更および取得することができます。

動的プロキシ

*動的プロキシ*とは、**Service** で提供される **getPort** メソッドの一つを使用するクライアントプロキシのインスタンスです。**portName** は、サービスが使用する WSDL ポートの名前を指定します。**serviceEndpointInterface** は、作成された動的プロキシインスタンスのサポートするサービスエンドポイントインターフェースを指定します。

例13.21 getPort メソッド

```
public <T> T getPort(QName portName, Class<T> serviceEndpointInterface)
public <T> T getPort(Class>T< serviceEndpointInterface)
```

*サービスエンドポイントインターフェース*は通常 **wsconsume.sh** コマンドを使用して生成されます。これにより WSDL が解析されて、Java クラスが作成されます。

ポートを返す、型指定されたメソッドも提供されます。このようなメソッドは、SEI を実装する動的プロキシも返します。以下の例を参照してください。

例13.22 サービスポートの戻り値

```
@WebServiceClient(name = "TestEndpointService", targetNamespace =
"http://org.jboss.ws/wsref",
   wsdlLocation = "http://localhost.localdomain:8080/jaxws-samples-
webserviceref?wsdl")

public class TestEndpointService extends Service
{
    ...

    public TestEndpointService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    @WebEndpoint(name = "TestEndpointPort")
    public TestEndpoint getTestEndpointPort()
    {
        return (TestEndpoint)super.getPort(TESTENDPOINTPORT,
TestEndpoint.class);
    }
}
```

@WebServiceRef

@WebServiceRef アノテーションは Web サービスの参照を宣言します。これは http://www.jcp.org/en/jsr/summary?id=250 で定義されている **javax.annotation.Resource** アノテーションにより示されるリソースパターンに従います。

@WebServiceRef のユースケース

- このアノテーションは、生成された **Service** クラス型である参照を定義するために使用することができます。この場合、種類要素と値要素はそれぞれ生成された **Service** クラス型を参照します。また、アノテーションが適用されるフィールドまたはメソッドの宣言によって参照型を推定することができる場合、種類要素および値要素にデフォルト値の **Object.class** を使用することができますが、必須ではありません。型が推測できない場合には、少なくとも種類要素は非デフォルト値で示す必要があります。

- このアノテーションを使用して型が SEI の参照を定義することができます。この場合、アノテーションが設定されたフィールドまたはメソッドの宣言から参照型を推定することができるならば、種類要素にデフォルト値を使用することができますが、必須ではありません 。ただし、値要素には常に、生成されたサービスクラス型を使用する必要があります。これは、**javax.xml.ws.Service** のサブタイプです。**wsdlLocation** 要素がある場合には、参照対象の生成されたサービスクラスの **@WebService** アノテーションで指定された WSDL ロケーション情報をオーバーライドします。

  例13.23 @WebServiceRef の例

  ```
  public class EJB3Client implements EJB3Remote
  {
     @WebServiceRef
     public TestEndpointService service4;
  ```

```
@WebServiceRef
public TestEndpoint port3;
```

Dispatch

XML Web Services は、Java EE コンテナ内にデプロイされたエンドポイントと任意のクライアントとの間における通信に XML メッセージを使用します。XML メッセージでは *Simple Object Access Protocol (SOAP)* と呼ばれる XML 言語を採用しています。JAX-WS API は、エンドポイントとクライアントがそれぞれ SOAP メッセージを送受信し、SOAP メッセージから Java への変換およびその逆の変換を行うことを可能にするメカニズムを提供します。これは **marshalling** および **unmarshalling** と呼ばれています。

場合によっては、変換の結果ではなく、raw SOAP メッセージ自体にアクセスする必要があります。**Dispatch** クラスはこの機能を提供します。**Dispatch** は 2 つの使用モードで動作し、次にあげる定数のいずれか一方により特定されます。

- **javax.xml.ws.Service.Mode.MESSAGE** - このモードは、クライアントアプリケーションがプロトコル固有のメッセージ構造を使用して直接連動するように指示します。SOAP プロトコルバインディングと併用すると、クライアントアプリケーションは SOAP メッセージと直接連動します。

- **javax.xml.ws.Service.Mode.PAYLOAD** - このモードを使用すると、クライアントはペイロード自体と連動します。たとえば、 SOAP プロトコルバインディングと併用した場合、クライアントアプリケーションは SOAP メッセージ全体ではなく、SOAP ボディのコンテンツと連動します。

Dispatch は、メッセージまたはペイロードを XML として構築する必要がある低レベルの API で、個別のプロトコルおよびメッセージまたはペイロード構造の詳細知識の標準に準拠します。**Dispatch** は、あらゆるタイプのメッセージまたはメッセージペイロードの入出力をサポートする、汎用クラスです。

例13.24 Dispatch の使用法

```
Service service = Service.create(wsdlURL, serviceName);
Dispatch dispatch = service.createDispatch(portName, StreamSource.class,
Mode.PAYLOAD);

String payload = "<ns1:ping
xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
dispatch.invokeOneWay(new StreamSource(new StringReader(payload)));

payload = "<ns1:feedback
xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
Source retObj = (Source)dispatch.invoke(new StreamSource(new
StringReader(payload)));
```

非同期呼び出し

BindingProvider インターフェースはクライアントが使用可能なプロトコルバインディングを提供するコンポーネントを表します。これはプロキシによって実装され、**Dispatch** インターフェースによって拡張されます。

BindingProvider インスタンスは非同期オペレーション機能を提供することが可能です。非同期オペレーション呼び出しは、呼び出し時に **BindingProvider** インスタンスから切り離されます。オペレーション完了時には、応答コンテキストは更新されず、その代わりに**Response** インターフェースを使用して別の応答コンテキストを利用できるようになります。

例13.25 非同期呼び出しの例

```
public void testInvokeAsync() throws Exception
{
   URL wsdlURL = new URL("http://" + getServerHost() + ":8080/jaxws-
samples-asynchronous?wsdl");
   QName serviceName = new QName(targetNS, "TestEndpointService");
   Service service = Service.create(wsdlURL, serviceName);
   TestEndpoint port = service.getPort(TestEndpoint.class);
   Response response = port.echoAsync("Async");
   // access future
   String retStr = (String) response.get();
   assertEquals("Async", retStr);
}
```

@Oneway 呼び出し

@Oneway アノテーションは、所定の **Web** メソッドが入力メッセージを受け取っても出力メッセージは返さないことを表します。通常、**@Oneway** メソッドは、ビジネスメソッドが実行される前に、制御のスレッドを呼び出し元アプリケーションに返します。

例13.26 @Oneway 呼び出しの例

```
@WebService (name="PingEndpoint")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class PingEndpointImpl
{
   private static String feedback;

   @WebMethod
   @Oneway
   public void ping()
   {
      log.info("ping");
      feedback = "ok";
   }

   @WebMethod
   public String feedback()
   {
      log.info("feedback");
      return feedback;
   }
}
```

タイムアウトの設定

HTTP 接続のタイムアウトの動作およびメッセージの受信を待つクライアントのタイムアウトは 2 つの

異なるプロパティによって制御されます。第1のプロパティは **javax.xml.ws.client.connectionTimeout**、第2のプロパティは **javax.xml.ws.client.receiveTimeout** です。各プロパティはミリ秒で表します。正しい構文は次のとおりです。

例13.27 JAX-WS タイムアウト設定

```
public void testConfigureTimeout() throws Exception
{
   //Set timeout until a connection is established

((BindingProvider)port).getRequestContext().put("javax.xml.ws.client.con
nectionTimeout", "6000");

   //Set timeout until the response is received
   ((BindingProvider)
port).getRequestContext().put("javax.xml.ws.client.receiveTimeout",
"1000");

   port.echo("testTimeout");
}
```

バグを報告する

13.5. JAX-WS 開発に関する参考資料

13.5.1. Web Services Addressing (WS-Addressing) の有効化

要件

- お使いのアプリケーションに既存の JAX-WS サービスとクライアント設定がなければなりません。

手順13.1 クライアントコードのアノテートおよび更新

1. **サービスエンドポイントのアノテーション**
   アプリケーションのエンドポイントコードに **@Addressing** アノテーションを追加します。

   例13.28 @Addressing アノテーション

   このサンプルでは、通常の JAX-WS エンドポイントに**@Addressing** アノテーションを追加する場合です。

   ```
   package org.jboss.test.ws.jaxws.samples.wsa;

   import javax.jws.WebService;
   import javax.xml.ws.soap.Addressing;

   @WebService
   (
      portName = "AddressingServicePort",
      serviceName = "AddressingService",
   ```

```
   wsdlLocation = "WEB-INF/wsdl/AddressingService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-
extensions/wsaddressing",
   endpointInterface =
"org.jboss.test.ws.jaxws.samples.wsa.ServiceIface"
)
@Addressing(enabled=true, required=true)
public class ServiceImpl implements ServiceIface
{
   public String sayHello()
   {
      return "Hello World!";
   }
}
```

2. **クライアントコードの更新**
   アプリケーションでクライアントコードを更新し **WS-Addressing** を設定

例13.29 WS-Addressing のクライアント設定

このサンプルでは、通常の **JAX-WS** クライアントを更新し **WS-Addressing** を設定しています。

```
package org.jboss.test.ws.jaxws.samples.wsa;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.soap.AddressingFeature;

public final class AddressingTestCase
{
   private final String serviceURL =
       "http://localhost:8080/jaxws-samples-
wsa/AddressingService";

   public static void main(String[] args) throws Exception
   {
      // construct proxy
      QName serviceName =
          new QName("http://www.jboss.org/jbossws/ws-
extensions/wsaddressing",
                    "AddressingService");
      URL wsdlURL = new URL(serviceURL + "?wsdl");
      Service service = Service.create(wsdlURL, serviceName);
      ServiceIface proxy =
          (ServiceIface)service.getPort(ServiceIface.class,
                                        new
AddressingFeature());
      // invoke method
      proxy.sayHello();
   }
}
```

結果

クライアントとエンドポイントは WS-Addressing を使い通信を行うようになりました。

バグを報告する

13.5.2. JAX-WS の共通 API リファレンス

一部の JAX-WS 開発概念は Web サービスエンドポイントとクライアントの間で共有されます。これには、ハンドラーフレームワーク、メッセージコンテキスト、フォルトハンドリングなどが含まれます。

ハンドラーフレームワーク

ハンドラーフレームワークは JAX-WS プロトコルバインディングにより、サーバーコンポーネントであるクライアントおよびエンドポイントのランタイムに実装されます。プロキシおよび **Dispatch** のインスタンスは、*バインディングプロバイダー*と総称されており、それぞれがプロトコルバインディングを使用して抽象機能を特定のプロトコルにバインドします。

クライアントおよびサーバー側のハンドラーは *ハンドラーチェーン*として知られる順序付きリストにまとめられます。ハンドラーチェーン内のハンドラーは、メッセージが送受信されるごとに呼び出されます。受信メッセージは、バインディングプロバイダーが処理する前にハンドラーによって処理されます。送信メッセージはバインディングプロバイダーが処理した後にハンドラーによって処理されます。

ハンドラーはメッセージコンテキストとともに呼び出されます。これは、受信メッセージと送信メッセージにアクセスして変更し、プロパティセットを管理するメソッドを提供します。メッセージコンテキストのプロパティは個々のハンドラー間およびハンドラー/クライアント/サービス実装間における通信を円滑化します。ハンドラーのタイプによって、一緒に呼び出されるメッセージコンテキストのタイプが異なります。

メッセージハンドラーのタイプ

論理ハンドラー

*論理ハンドラー*はメッセージコンテキストプロパティーおよびメッセージペイロードでのみ動作します。論理ハンドラーはプロトコル非依存なので、メッセージのプロトコル固有の部分に影響を与えることはできません。論理ハンドラーはインターフェース **`javax.xml.ws.handler.LogicalHandler`** を実装します。

プロトコルハンドラー

Protocol handlers はメッセージコンテキストおよびプロトコル固有のメッセージでのみ動作します。プロトコルハンドラーは特定のプロトコルに固有で、プロトコル固有のメッセージアスペクトにアクセスし、変更することが可能です。 プロトコルハンドラーは **`javax.xml.ws.handler.Handler except javax.xml.ws.handler.LogicalHandler`** から派生する任意のインターフェースを実装します。

サービスエンドポイントハンドラー

サービスエンドポイントでは、ハンドラーは **`@HandlerChain`** アノテーションを使用して定義されます。ハンドラーチェーンファイルのロケーションは、**`externalForm`** 内の絶対 **`java.net.URL`**、あるいはソースファイル/クラスファイルからの相対パスで指定することができます。

例13.30 サービスエンドポイントハンドラーの例

```
@WebService
@HandlerChain(file = "jaxws-server-source-handlers.xml")
public class SOAPEndpointSourceImpl
```

```
{
    ...
}
```

サービスクライアントハンドラー

JAX-WS クライアントでは、ハンドラーは サービスエンドポイント同様に **@HandlerChain** アノテーションを使用するか、動的に JAX-WS API を使用して定義します。

例13.31 API を使用したサービスクライアントハンドラーの定義

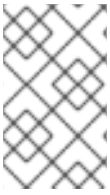
```
Service service = Service.create(wsdlURL, serviceName);
Endpoint port = (Endpoint)service.getPort(Endpoint.class);

BindingProvider bindingProvider = (BindingProvider)port;
List<Handler> handlerChain = new ArrayList<Handler>();
handlerChain.add(new LogHandler());
handlerChain.add(new AuthorizationHandler());
handlerChain.add(new RoutingHandler());
bindingProvider.getBinding().setHandlerChain(handlerChain);
```

setHandlerChain メソッドへのコールが必要です。

メッセージコンテキスト

MessageContext インターフェースは、全 JAX-WS メッセージコンテキスト用のスーパーインターフェースです。追加のメソッドと定数を使用して **Map<String,Object>** を拡張し、ハンドラーチェーン内のハンドラーが処理関連の状態を共有できるようにするプロパティセットを管理します。たとえば、ハンドラーは **put** メソッドを使用してメッセージコンテキストにプロパティを挿入することができます。その後、ハンドラーチェーン内の単一または複数のハンドラーは、 **get** メソッドでメッセージを取得できるようになります。

プロパティは、**APPLICATION** または **HANDLER** としてスコープ指定されます。すべてのプロパティは、特定のエンドポイントの *メッセージ交換パターン (MEP)* のインスタンスの全ハンドラーが使用することができます。たとえば、論理ハンドラーがメッセージコンテキストにプロパティを挿入すると、そのプロパティは、MET インスタンスの実行中にチェーン内の任意のプロトコルハンドラーも使用することができます。



注記

非同期メッセージ交換パターン (MEP) により、HTTP 接続レベルでのメッセージの非同期的な送受信が可能となります。これは、要求コンテキストに追加のプロパティーを設定することによって有効にできます。

APPLICATION レベルにスコープ指定されているプロパティはクライアントアプリケーションとサービスエンドポイントの実装でも使用することができます。プロパティの **defaultscope** は **HANDLER** です。

論理メッセージと SOAP メッセージでは使用するコンテキストが異なります。

論理メッセージコンテキスト

論理ハンドラーが呼び出される際には、タイプ **LogicalMessageContext** のメッセージコンテキストを受信します。**LogicalMessageContext** は、メッセージペイロードを取得/変更するメソッドを使用して **MessageContext** を拡張します。これは、メッセージのプロトコル固有のアスペクトへのアクセスは提供しません。プロトコルバインディングは、論理メッセージコンテキストを介して使用可能なメッセージコンポーネントを定義します。SOAP バインディングにデプロイされている論理ハンドラーは、SOAP ボディーのコンテンツにアクセス可能ですが、SOAP ヘッダーにはアクセスできません。一方、XML/HTTP バインディングは論理ハンドラーがメッセージの XML ペイロード全体にアクセスできることを定義します。

SOAP メッセージコンテキスト

SOAP ハンドラーが呼び出される際には、**SOAPMessageContext** を受信します。**SOAPMessageContext** は SOAP メッセージペイロードを取得/変更するメソッドを使用して **MessageContext** を拡張します。

フォルドハンドリング

アプリケーションは **SOAPFaultException** またはアプリケーション固有のユーザー例外をスローします。後者の場合、必要とされる障害ラッパー (fault wrapper) Bean が既にデプロイメントの一部となっていなければ、ランタイムに生成されます。

例13.32 フォルトハンドリングの例

```
public void throwSoapFaultException()
{
   SOAPFactory factory = SOAPFactory.newInstance();
   SOAPFault fault = factory.createFault("this is a fault string!", new
QName("http://foo", "FooCode"));
   fault.setFaultActor("mr.actor");
   fault.addDetail().addChildElement("test");
   throw new SOAPFaultException(fault);
}
```

```
public void throwApplicationException() throws UserException
{
   throw new UserException("validation", 123, "Some validation error");
}
```

JAX-WS アノテーション

JAX-WS API で使用可能なアノテーションは JSR-224 で定義されています。この定義は http://www.jcp.org/en/jsr/detail?id=224 に記載されています。これらのアノテーションは **javax.xml.ws** パッケージに含まれています。

JWS API で使用できるアノテーションは、 JSR-181 で定義されています。この定義は http://www.jcp.org/en/jsr/detail?id=181 に記載されています。これらのアノテーションは**javax.jws** パッケージに含まれています。

バグを報告する

第14章 アプリケーション内のアイデンティティー

14.1. 基本概念

14.1.1. 暗号化について

*暗号化*とは、数学的なアルゴリズムを適用して機密情報を分かりにくくすることを言います。暗号化はデータの侵害やシステム機能の停止などのリスクからインフラストラクチャーを保護する基盤の 1 つとなります。

暗号化はパスワードなどの簡単な文字列データへ適用することができます。また、データ通信のストリームへ適用することも可能です。例えば、HTTPS プロトコルはデータを転送する前にすべてのデータを暗号化します。セキュアシェル (SSH) プロトコルを使用して 1 つのサーバーから別のサーバーへ接続する場合、すべての通信が暗号化された *トンネル*で送信されます。

バグを報告する

14.1.2. セキュリティードメインについて

セキュリティードメインは、JBoss Enterprise Application Platform のセキュリティーサブシステムの一部になります。セキュリティー設定は、管理対象ドメインの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よって集中管理されるようになりました。

セキュリティードメインは認証、承認、セキュリティーマッピング、監査の設定によって構成されます。セキュリティードメインは *Java Authentication and Authorization Service (JAAS)*の宣言的セキュリティーを実装します。

認証とはユーザーアイデンティティーを検証することを言います。セキュリティー用語では、このユーザーを *プリンシパル*と呼びます。認証と承認は異なりますが、含まれている認証モジュールの多くは承認の処理も行います。

*承認*とは、許可または禁止されている動作に関する情報が含まれるセキュリティーポリシーのことです。セキュリティー用語では、ロールと呼ばれます。

*セキュリティーマッピング*とは、情報をアプリケーションに渡す前にプリンシパル、ロール、または属性から情報を追加、編集、削除する機能のことです。

監査マネージャーを使用すると、*プロバイダーモジュール*を設定してセキュリティーイベントの報告方法を制御できます。

セキュリティードメインを使用する場合、アプリケーション自体から特定のセキュリティー設定をすべて削除することが可能です。これにより、一元的にセキュリティーパラメーターを変更できるようにします。このような設定構造が有効な一般的な例には、アプリケーションをテスト環境と実稼動環境間で移動するプロセスがあります。

バグを報告する

14.1.3. SSL 暗号化について

SSL (Secure Socket Layer) は、2 つのシステム間のネットワークトラフィックを暗号化します。接続の*ハンドシェーク*フェーズ中に生成され、2 つのシステムのみが認識する共通鍵を使用して、2 つのシステム間のトラフィックが暗号化されます。

共通鍵をセキュアに交換するため、SSL は PKI (Public Key Infrastructure) を利用します。PKI とは*キーペア*を用いる暗号化の方法です。キーペアは公開鍵と秘密鍵の 2 つのペアの鍵で構成されま

す。公開鍵は他のユーザーと共有され、データの暗号化に使用されます。秘密鍵は公開されず、公開鍵で暗号化されたデータを復号化する時に使用されます。

クライアントが安全な接続を要求した場合、安全な通信が開始される前にハンドシェイクフェーズが実行されます。SSL のハンドシェイク中にサーバーは公開鍵を証明書としてクライアントに渡します。この証明書にはサーバーの ID (サーバーの URL)、サーバーの公開鍵、証明書を認証するデジタル署名が含まれています。その後、クライアントは証明書を検証し、この証明書が信頼できるものかを判断します。この証明書を信頼する場合、クライアントは共通鍵を SSL 接続に対して生成し、サーバーの公開鍵を使用して暗号化してからサーバーに戻します。サーバーは秘密鍵を使用して、共通鍵を復号化します。その後、同じ接続でこれらの 2 つのマシンが行う通信はこの共通鍵を使い暗号化されます。

バグを報告する

14.1.4. 宣言的セキュリティーについて

*宣言的セキュリティー*とは、セキュリティー管理にコンテナを使うことで、お使いのアプリケーションコードからセキュリティーの問題を切り離す方法です。コンテナにより、ファイルのパーミッション、またはユーザー、グループ、ロールに基づき承認を行います。このアプローチは、セキュリティー関連すべてをアプリケーション自体で請け負う *プログラム的*セキュリティーよりも優れています。

JBoss Enterprise Application Platform はセキュリティードメインより宣言的セキュリティーを提供します。

バグを報告する

14.2. アプリケーションのロールベースセキュリティー

14.2.1. アプリケーションセキュリティーについて

アプリケーションの開発者はアプリケーションをセキュアにすることが多面的で重要であることを認識しています。JBoss Enterprise Application Platform は以下のような機能が含まれる、セキュアなアプリケーションの作成に必要なツールをすべて提供します。

- 「認証について」
- 「承認について」
- 「セキュリティー監査について」
- 「セキュリティーマッピングについて」
- 「宣言的セキュリティーについて」
- 「EJB メソッドパーミッションについて」
- 「EJB セキュリティーアノテーションについて」

「アプリケーションでのセキュリティードメインの使用」も参照してください。

バグを報告する

14.2.2. 認証について

認証とは、サブジェクトを識別し、身分が本物であるかを検証することを言います。最も一般的な認証メカニズムはユーザー名とパスワードの組み合わせです。その他の一般的な認証メカニズムは共有

キー、スマートカード、または指紋などを使用します。 Java Enterprise Edition の宣言的セキュリティーでは、成功した認証の結果のことをプリンシパルと呼びます。

JBoss Enterprise Application Platform は認証モジュールのプラグ可能なシステムを使用して、組織ですでに使用している認証システムへ柔軟に対応し、統合を実現します。各セキュリティードメインには 1 つ以上の設定された認証モジュールが含まれます。各モジュールには、挙動をカスタマイズするための追加の設定パラメーターが含まれています。Web ベースの管理コンソール内に認証サブシステムを設定するのが最も簡単な方法です。

認証と承認が関連している場合も多くありますが、認証と承認は同じではありません。含まれている多くの認証モジュールは承認も処理します。

バグを報告する

14.2.3. 承認について

承認とはアイデンティティーを基にリソースへのアクセスを許可または拒否するメカニズムのことです。プリンシパルに提供できる宣言的セキュリティーロールのセットとして実装されます。

JBoss Enterprise Application Platform はモジュラーシステムを使用して承認を設定します。各セキュリティードメインに 1 つ以上の承認ポリシーが含まれるようにすることができます。各ポリシーには動作を定義する基本モジュールがあり、特定のフラグや属性より設定されます。Web ベースの管理コンソールを使用すると承認サブシステムを最も簡単に設定できます。

承認は認証とは異なり、通常は認証後に承認が行われます。認証モジュールの多くは承認も処理します。

バグを報告する

14.2.4. セキュリティー監査について

セキュリティー監査とは、セキュリティーサブシステム内で発生したイベントに応答するため、ログへの書き込みなどのイベントをトリガーすることです。監査のメカニズムは、認証、承認、およびセキュリティーマッピングの詳細と共に、セキュリティードメインの一部として設定されます。

監査には *プロバイダーモジュール*が使用されます。含まれているプロバイダーモジュールを使用するか、独自のモジュールを実装することができます。

バグを報告する

14.2.5. セキュリティーマッピングについて

セキュリティーマッピングを使用すると、認証または承認が実行された後、情報がアプリケーションに渡される前に認証情報と承認情報を組み合わせることができます。この例の 1 つが、X509 証明書を認証に使用した後、プリンシパルを証明書からアプリケーションが表示できる論理名へ変換することです。

プリンシパル (認証) やロール (承認)、証明情報 (プリンシパルやロールでない属性) をマッピングすることが可能です。

ロールマッピングは、認証後にサブジェクトへロールを追加、置換、または削除するために使用されます。

プリンシパルマッピングは、認証後にプリンシパルを変更するために使用されます。

属性マッピングは、外部システムからの属性値をアプリケーションで使用するために変換したり、逆にそのような外部システムへ属性を変換したりするために使用されます。

バグを報告する

14.2.6. セキュリティー拡張アーキテクチャーについて

JBoss Enterprise Application Platform のセキュリティー拡張のアーキテクチャーは 3 つの部分で構成されています。基盤のセキュリティーインフラストラクチャーが LDAP や Kerberos、その他の外部システムであるかに関わらず、これらの 3 つの部分はアプリケーションを基盤のセキュリティーインフラストラクチャーへ接続します。

JAAS

インフラストラクチャーの最初の部分は JAAS API になります。JAAS はセキュリティーインフラストラクチャーとアプリケーションの間の抽象化レイヤーを提供するプラグイン可能なフレームワークです。

JAAS の主な実装は、**AuthenticationManager** インターフェースと **RealmMapping** インターフェースを実装する **org.jboss.security.plugins.JaasSecurityManager** です。**JaasSecurityManager** は、対応するコンポーネントデプロイメント記述子の**<security-domain>** 要素を基に、EJB レイヤーと Web コンテナレイヤーに統合します。

JAAS に関する詳細は 「Java Authentication and Authorization Service (JAAS)」 を参照してください。

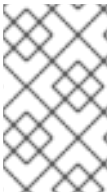
JaasSecurityManagerService MBean

JaasSecurityManagerService MBean サービスはセキュリティーマネージャーを管理します。名前は JAAS で始まりますが、処理するセキュリティーマネージャーは実装で JAAS を使用する必要はありません。この名前は、デフォルトのセキュリティーマネージャー実装が **JaasSecurityManager** であることを示しています。

JaasSecurityManagerService の主要な役割はセキュリティーマネージャー実装を外部化することです。**AuthenticationManager** インターフェースと **RealmMapping** インターフェースの代替の実装を提供してセキュリティーマネージャーの実装を変更することができます。

JaasSecurityManagerService の 2 つ目の基礎的な役割は、JNDI **javax.naming.spi.ObjectFactory** 実装を提供して JNDI 名とセキュリティーマネージャー実装との間でバインディングの簡単なコードのない管理を実現することです。セキュリティーを有効にするには、**<security-domain>** デプロイメント記述子要素よりセキュリティーマネージャー実装の JNDI 名を指定します。

JNDI 名を指定する時、オブジェクトバインディングが既に存在する必要があります。JNDI 名とセキュリティーマネージャー間のバインディング設定を簡単にするため、**JaasSecurityManagerService** が *次のネーミングシステムリファレンス*をバインドし、**java:/jaas** という名前の JNDI の **ObjectFactory** として **JaasSecurityManagerService** 自体をノミネートします。これにより、**java:/jaas/*XYZ*** という形式の命名規則を **<security-domain>**要素の値とすることができます。セキュリティードメインの名前を取るコンストラクターを使用して **SecurityManagerClassName** 属性によって指定されるクラスのインスタンスを作成して、***XYZ*** セキュリティードメインのセキュリティーマネージャーインスタンスは必要な時に作成されます。



注記

java:/jaas プレフィックスがデプロイメント記述子に含まれるようにする必要はありません。後方互換性を維持するため指定することがあるかもしれませんが、このプレフィックスは無視されます。

JaasSecurityDomain MBean

org.jboss.security.plugins.JaasSecurityDomain は、 SSL やその他の暗号化のユースケースをサポートするため **KeyStore** や **KeyManagerFactory**、**TrustManagerFactory** の概念を追加する **JaasSecurityManager** の拡張です。

詳細情報

詳細や動作しているセキュリティーアーキテクチャーの実例については 「Java Authentication and Authorization Service (JAAS) について」 を参照してください。

バグを報告する

14.2.7. Java Authentication and Authorization Service (JAAS)

*Java Authentication and Authorization Service (JAAS)*は、ユーザーの認証や承認向けに設計された Java パッケージで構成されるセキュリティー API です。API は標準的なプラグ可能認証モジュール (PAM) フレームワークの Java 実装です。Java Enterprise Edition のアクセス制御アーキテクチャーを拡張し、ユーザーベースの承認をサポートします。

JBoss Enterprise Application Platform では JAAS は宣言的ロールベースセキュリティーのみを提供します。宣言的セキュリティーについての詳細は 「宣言的セキュリティーについて」 を参照してください。

JAAS は Kerberos や LDAP などの基礎となる認証技術から独立しています。アプリケーションを変更せずに、JAAS の設定を変更するだけで基礎となるセキュリティー構造を変更することが可能です。

バグを報告する

14.2.8. Java Authentication and Authorization Service (JAAS) について

JBoss Enterprise Application Platform 6 のセキュリティーアーキテクチャーは、セキュリティー設定サブシステムと、アプリケーション内の複数の設定ファイルに含まれるアプリケーション固有のセキュリティー設定、MBean として実装される JAAS セキュリティーマネージャーで構成されます。

ドメイン、サーバーグループ、サーバー固有の設定

サーバーグループ (管理対象ドメイン内) とサーバー (スタンドアロンサーバー内) にはセキュリティードメインの設定が含まれます。セキュリティードメインには、認証、承認、マッピング、監査のモジュールの組み合わせと設定詳細に関する情報が含まれています。アプリケーションは必要なセキュリティードメインを名前で **jboss-web.xml** に指定します。

アプリケーション固有の設定

アプリケーション固有の設定は次の 4 つのファイルの 1 つ以上に設定されます。

表14.1 アプリケーション固有の設定ファイル

ファイル	説明
ejb-jar.xml	EJB の **META-INF** ディレクトリにある Enterprise JavaBean (EJB) アプリケーションのデプロイメント記述子です。**ejb-jar.xml** を使用してロールを指定し、アプリケーションレベルでプリンシパルへマッピングします。また、特定のメソッドやクラスを特定のロールへ制限することも可能です。セキュリティーに関係しない他の EJB 固有の設定に対しても使用できます。
web.xml	Java Enterprise Edition (EE) の Web アプリケーションのデプロイメント記述子です。**web.xml** を使用して、認証や承認にアプリケーションが使用するセキュリティードメインを宣言します。また、許可される HTTP リクエストのタイプを制限するなど、アプリケーションのリソースやトランスポートを制約するため使用することもできます。このファイルに簡単な Web ベースの認証を設定することもできます。セキュリティーに関係しない他のアプリケーション固有の設定に使用することもできます。
jboss-ejb3.xml	**ejb-jar.xml** 記述子への JBoss 固有の拡張が含まれます。
jboss-web.xml	**web.xml** 記述子への JBoss 固有の拡張が含まれます。

注記

ejb-jar.xml と **web.xml** は Java Enterprise Edition (Java EE) 仕様に定義されています。**jboss-ejb3.xml** は **ejb-jar.xml** の JBoss 固有の拡張を提供し、**jboss-web.xml** は **web.xml** の JBoss 固有の拡張を提供します。

JAAS セキュリティーマネージャー MBean

*Java Authentication and Authorization Service (JAAS)*はプラグ可能認証モジュール (PAM) を使用した、Java アプリケーションのユーザーレベルのセキュリティーに対するフレームワークです。JAAS は Java ランタイム環境 (JRE) に統合されます。JBoss Enterprise Application Platform では、コンテナ側のコンポーネントは **org.jboss.security.plugins.JaasSecurityManager** MBean で、**AuthenticationManager** インターフェースと **RealmMapping** インターフェースのデフォルト実装を提供します。

JaasSecurityManager MBean はアプリケーションの EJB または Web デプロイメント記述子ファイルに指定されているセキュリティードメインを EJB および Web コンテナレイヤーに統合します。アプリケーションがデプロイすると、コンテナはデプロイメント記述子に指定されたセキュリティードメインをコンテナのセキュリティーマネージャーインスタンスへ関連付けします。セキュリティーマネージャーはセキュリティードメインをサーバーグループまたはスタンドアローンサーバー上に設定します。

クライアントと JAAS を持つコンテナとの間の対話フロー

JaasSecurityManager は JAAS パッケージを使用して AuthenticationManager と RealmMapping インターフェースの動作を実装します。JaasSecurityManager へ割り当てられたセキュリティードメインに設定されたログインモジュールインスタンスを実行するとこの動作が生じます。ログインモジュール

はセキュリティードメインのプリンシパルの認証やロールマッピングの挙動を実装します。ドメインの異なるログインモジュール設定を組み込むと、異なるセキュリティードメイン全体で JaasSecurityManager を使用することができます。

JaasSecurityManager がどのように JAAS 認証プロセスを使用するかを説明する次の手順を見てください。この手順はメソッド **EJBHome** を実装するメソッドのクライアント呼び出しの概要になります。EJB は既にサーバーにデプロイされ、**EJBHome** インターフェースメソッドは **ejb-jar.xml** 記述子の <method-permission> 要素を使用してセキュアな状態になっています。**jboss-ejb3.xml** ファイルの <security-domain> 要素に指定される **jwdomain** セキュリティードメインを使用します。以下の図は後で説明する手順を表しています。

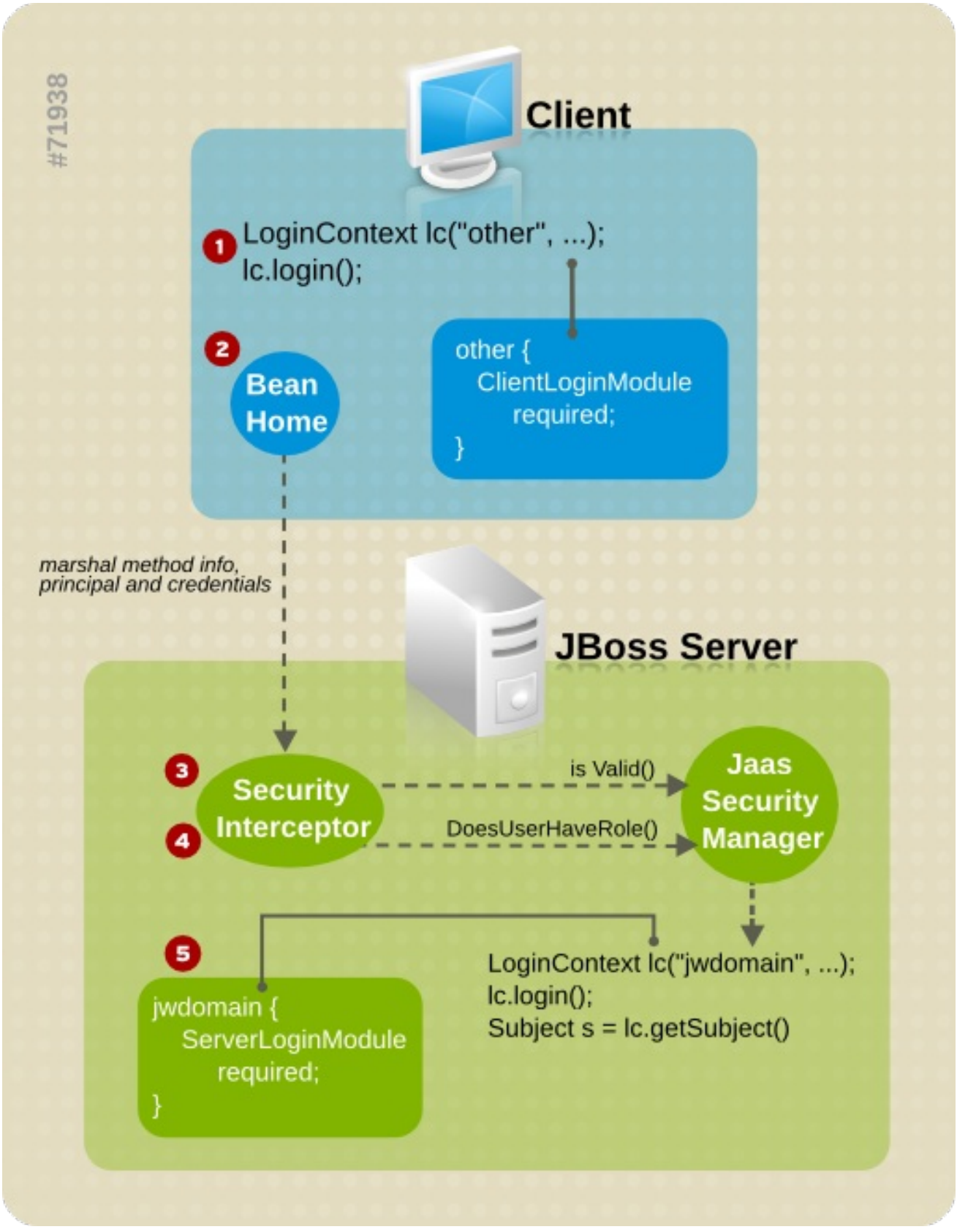


図14.1 保護された EJB メソッド呼び出しの手順

1. クライアントが JAAS のログインを実行し、認証のプリンシパルと認証情報を確立します。上図では **`Client Side Login`** とラベル付けされます。JNDI より実行することも可能です。

   JAAS ログインを実行するには、**LoginContext** インスタンスを作成し、使用する設定の名前を渡します。ここでの設定名は **`other`** になります。このワンタイムログインは、ログインプリンシパルと認証情報を後続の EJB メソッド呼び出しすべてへ関連付けます。プロセスがユーザーを認証するとは限りません。クライアント側のログインの性質は、クライアントが使用するロ

グインモジュール設定によって異なります。この例では、**other** というクライアント側ログイン設定エントリーが **ClientLoginModule** ログインモジュールを使用します。サーバー上で後で認証が行われるため、このモジュールはユーザー名とパスワードを EJB 呼び出しレイヤーへバインドします。クライアントのアイデンティティーはクライアント上で認証されません。

2. クライアントは **EJBHome** メソッドを取得し、このメソッドをサーバー上で呼び出します。呼び出しにはクライアントによって渡されたメソッド引数や、クライアント側 JAAS ログインからのユーザー ID や認証情報が含まれます。

3. サーバー上では、セキュリティーインターセプターがメソッドを呼び出したユーザーを認証します。これには別の JAAS ログインが関係します。

4. セキュリティードメインはログインモジュールの選択を決定します。セキュリティードメインの名前はログイン設定エントリー名として **LoginContext** コンストラクターへ渡されます。EJB セキュリティードメインは **jwdomain** です。JAAS 認証に成功すると、JAAS サブジェクトが作成されます。JAAS サブジェクトには次の詳細を含む PrincipalSet が含まれます。

    - デプロイメントセキュリティー環境よりクライアントアイデンティティーへ対応する **java.security.Principal** インスタンス。

    - ユーザーのアプリケーションドメインからのロール名が含まれる **Roles** と呼ばれる **java.security.acl.Group**。**org.jboss.security.SimplePrincipal** タイプのオブジェクトはロール名を表します。これらのロールは、**ejb-jar.xml** と **EJBContext.isCallerInRole(String)** メソッド実装の制約に従って EJB メソッドへのアクセスを検証します。

    - アプリケーションドメインの呼び出し側のアイデンティティーに対応する 1 つの **org.jboss.security.SimplePrincipal** が含まれる **CallerPrincipal** という名前の任意の **java.security.acl.Group**。CallerPrincipal グループメンバーは **EJBContext.getCallerPrincipal()** メソッドによって返される値です。このマッピングは、運用セキュリティー環境のプリンシパルがアプリケーションが認識するプリンシパルへマッピングできるようにします。CallerPrincipal マッピングが存在しない場合、運用プリンシパルはアプリケーションドメインプリンシパルと同じになります。

5. EJB メソッドを呼び出しているユーザーは呼び出しが許可されているユーザーであることをサーバーが検証します。次の手順でこの承認を実行します。

    - EJB コンテナから EJBメソッドへアクセスすることが許可されるロールの名前を取得します。呼び出されたメソッドが含まれるすべての <method-permission> 要素の **ejb-jar.xml** 記述子 <role-name> 要素によってロール名が判断されます。

    - 割り当てられたロールがなかったり、メソッドが exclude-list 要素に指定されている場合、メソッドへのアクセスは拒否されます。それ以外の場合は、セキュリティーインターセプターによってセキュリティーマネージャー上で **doesUserHaveRole** メソッドが呼び出され、呼び出し側に割り当てられたロール名の 1 つがあるかどうかを確認します。このメソッドはロール名より繰り返され、認証されたユーザーの **Subject Roles** グループに割り当てられたロール名を持つ SimplePrincipal が含まれるか確認します。Roles グループメンバーのロール名がある場合はアクセスが許可されます。メンバーのロール名がない場合はアクセスが拒否されます。

    - EJB がカスタムのセキュリティープロキシを使用する場合、メソッドの呼び出しはプロキシへ委譲されます。セキュリティープロキシが呼び出し側へのアクセスを拒否すると、**java.lang.SecurityException** がスローされます。それ以外の場合は EJB メソッドへのアクセスは許可され、メソッド呼び出しは次のコンテナインターセプターへ渡されます。SecurityProxyInterceptor はこのチェックを処理し、このインターセプターは表示されません。

- Web 接続要求の場合、**web.xml** で定義され、要求されたリソースとアクセスされた HTTP メソッドに一致するセキュリティー制約を Web サーバーがチェックします。

  要求に対して制約が存在する場合、Web サーバーは JaasSecurityManager を呼び出し、プリンシパルの認証を行います。これにより、確実にユーザーロールがプリンシパルオブジェクトへ関連付けられているようにします。

バグを報告する

14.2.9. アプリケーションでのセキュリティードメインの使用

概要

アプリケーションでセキュリティードメインを使用するには、最初にサーバーの設定ファイルまたはアプリケーションの記述子ファイルのいずれかにドメインを設定する必要があります。その後、使用する EJB に必要なアノテーションを追加する必要があります。ここでは、アプリケーションでセキュリティードメインを使用するために必要な手順について取り上げます。

手順14.1 セキュリティードメインを使用するようアプリケーションを設定

1. **セキュリティードメインの定義**
   セキュリティードメインは、サーバーの設定ファイルまたはアプリケーションの記述子ファイルのいずれかに定義できます。

   - **サーバーの設定ファイルへセキュリティードメインを設定**
     セキュリティードメインは、サーバーの設定ファイルの **security** サブシステムに設定されます。JBoss Enterprise Application Platform インスタンスが管理対象ドメインで実行されている場合、**domain/configuration/domain.xml** ファイルになります。JBoss Enterprise Application Platform インスタンスがスタンドアロンサーバーとして実行されている場合は **standalone/configuration/standalone.xml** ファイルになります。

     other、**jboss-web-policy** および **jboss-ejb-policy** セキュリティードメインはデフォルトとして JBoss Enterprise Application Platform 6 に提供されます。次の XML の例は、サーバーの設定ファイルの **security** サブシステムよりコピーされました。

```
<subsystem xmlns="urn:jboss:domain:security:1.2">
    <security-domains>
        <security-domain name="other" cache-type="default">
            <authentication>
                <login-module code="Remoting" flag="optional">
                    <module-option name="password-stacking"
value="useFirstPass"/>
                </login-module>
                <login-module code="RealmDirect"
flag="required">
                    <module-option name="password-stacking"
value="useFirstPass"/>
                </login-module>
            </authentication>
        </security-domain>
        <security-domain name="jboss-web-policy" cache-
type="default">
            <authorization>
                <policy-module code="Delegating"
flag="required"/>
            </authorization>
```

```
        </security-domain>
        <security-domain name="jboss-ejb-policy" cache-
type="default">
            <authorization>
                <policy-module code="Delegating"
flag="required"/>
            </authorization>
        </security-domain>
    </security-domains>
</subsystem>
```

管理コンソールまたは CLI を使用して、追加のセキュリティードメインを必要に応じて設定することができます。

- **アプリケーションの記述子ファイルにセキュリティードメインを設定**
  セキュリティードメインはアプリケーションの **WEB-INF/web.xml** ファイルにある **<jboss-web>** 要素の **<security-domain>** 子要素に指定されます。次の例は **my-domain** という名前のセキュリティードメインを設定します。

```
<jboss-web>
    <security-domain>my-domain</security-domain>
</jboss-web>
```

これが **WEB-INF/jboss-web.xml** 記述子に指定できる多くの設定の 1 つになります。

2. **EJB へ必要なアノテーションを追加**
   @SecurityDomain および **@RolesAllowed** アノテーションを使用してセキュリティーを EJB に設定します。次の EJBコードの例は、**guest** ロールのユーザーによる **other** セキュリティードメインへのアクセスを制限します。

```
package example.ejb3;

import java.security.Principal;

import javax.annotation.Resource;
import javax.annotation.security.RolesAllowed;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;

import org.jboss.ejb3.annotation.SecurityDomain;

/**
 * Simple secured EJB using EJB security annotations
 * Allow access to "other" security domain by users in a "guest"
role.
 */
@Stateless
@RolesAllowed({ "guest" })
@SecurityDomain("other")
public class SecuredEJB {

   // Inject the Session Context
```

```
    @Resource
    private SessionContext ctx;

    /**
     * Secured EJB method using security annotations
     */
    public String getSecurityInfo() {
       // Session context injected using the resource annotation
       Principal principal = ctx.getCallerPrincipal();
       return principal.toString();
    }
}
```

その他のコード例は、Red Hat カスタマーポータルより入手できる JBoss Enterprise Application Platform 6 Quickstarts バンドルの **ejb-security** クイックスタートを参照してください。

バグを報告する

14.2.10. サーブレットでのロールベースセキュリティーの使用

サーブレットにセキュリティーを追加するには、各サーブレットを URL パターンへマッピングし、保護する必要のある URL パターン上でセキュリティー制約を作成します。セキュリティー制約は、ロールに対して URL へのアクセスを制限します。認証と承認は WAR の **jboss-web.xml** に指定されたセキュリティードメインによって処理されます。

要件

サーブレットで ロールベースセキュリティーを使用する前に、アクセスの認証と承認に使用されるセキュリティードメインを JBoss Enterprise Application Platform のコンテナに設定する必要があります。

手順14.2 ロールベースセキュリティーのサーブレットへの追加

1. **サーブレットと URL パターンの間にマッピングを追加します。**
   web.xml の **<servlet-mapping>** 要素を使用して各サーブレットを URL パターンへマッピングします。次の例は **DisplayOpResult** と呼ばれるサーブレットを URL パターン **/DisplayOpResult** にマッピングします。

```
<servlet-mapping>
    <servlet-name>DisplayOpResult</servlet-name>
    <url-pattern>/DisplayOpResult</url-pattern>
</servlet-mapping>
```

2. **URL パターンにセキュリティー制約を追加します。**
   URL パターンをセキュリティー制約へマッピングするには、**<security-constraint>** を使用します。次の例は、URL パターン **/DisplayOpResult** のアクセスを、ロール **eap_admin** を持つプリンシパルのみに許可します。セキュリティードメインにロールが存在していなければなりません。

```
<security-constraint>
 <display-name>Restrict access to role eap_admin</display-name>
 <web-resource-collection>
```

```
  <web-resource-name>Restrict access to role eap_admin</web-
resource-name>
  <url-pattern>/DisplayOpResult/*</url-pattern>
 </web-resource-collection>
 <auth-constraint>
  <role-name>eap_admin</role-name>
 </auth-constraint>
 <security-role>
  <role-name>eap_admin</role-name>
 </security-role>
</security-constraint>
```

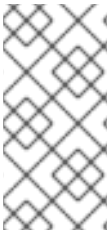
3. **WAR の `jboss-web.xml` にセキュリティードメインを指定します。**
   セキュリティードメインにサーブレットを接続するため、WAR の **`jboss-web.xml`** にセキュリティードメインを追加します。セキュリティードメインにはセキュリティー制約に対してプリンシパルを認証および承認する方法が設定されています。次の例は **`acme_domain`** というセキュリティードメインを使用します。

```
<jboss-web>
 ...
 <security-domain>acme_domain</security-domain>
 ...
</jboss-web>
```

バグを報告する

14.2.11. アプリケーションにおけるサードパーティー認証システムの使用

サードパーティーのセキュリティーシステムを JBoss Enterprise Application Platform に統合することができます。このようなシステムは通常トークンベースのシステムです。外部システムが認証を実行し、要求ヘッダーよりトークンを Web アプリケーションに返します。このような認証は*境界認証*と呼ばれることもあります。アプリケーションに境界認証を設定するには、カスタムの認証バルブを追加します。サードパーティープロバイダーのバルブがある場合はクラスパスに存在するようにし、以下の例とサードパーティー認証モジュールのドキュメントに従うようにしてください。



注記

JBoss Enterprise Application Platform 6 では設定するバルブの場所が変更になりました。**`context.xml`** デプロイメント記述子には設定されないようになりました。バルブは直接 **`jboss-web.xml`** 記述子に設定されます。**`context.xml`** は無視されるようになりました。

例14.1 基本的な認証バルブ

```
<jboss-web>
  <valve>
    <class-
name>org.jboss.security.negotiation.NegotiationAuthenticator</class-
name>
```

```
  </valve>
</jboss-web>
```

このバルブは Kerberos ベースの SSO に使用されます。また、Web アプリケーションに対してサードパーティーのオーセンティケーターを指定する最も単純なパターンを示しています。

例14.2 ヘッダー属性セットを持つカスタムバルブ

```
<jboss-web>
  <valve>
    <class-
name>org.jboss.web.tomcat.security.GenericHeaderAuthenticator</class-
name>
    <param>
      <param-name>httpHeaderForSSOAuth</param-name>
      <param-value>sm_ssoid,ct-remote-user,HTTP_OBLIX_UID</param-value>
    </param>
    <param>
      <param-name>sessionCookieForSSOAuth</param-name>
      <param-value>SMSESSION,CTSESSION,ObSSOCookie</param-value>
    </param>
  </valve>
</jboss-web>
```

この例ではバルブにカスタム属性を設定する方法が示されています。オーセンティケーターはヘッダー ID とセッション鍵の存在を確認し、ユーザー名とパスワードバルブとしてセキュリティー層を操作する JAAS フレームワークへ渡します。ユーザー名とパスワードの処理が可能で、サブジェクトに適切なロールを投入できるカスタムの JAAS ログインモジュールが必要となります。設定された値と一致するヘッダー値がない場合、通常のフォームベース認証のセマンティックが適用されます。

カスタムオーセンティケーターの作成

独自のオーセンティケーターの作成については本書の範囲外となりますが、次の Java コードが例として提供されています。

例14.3 GenericHeaderAuthenticator.java

```
/*
 * JBoss, Home of Professional Open Source.
 * Copyright 2006, Red Hat Middleware LLC, and individual contributors
 * as indicated by the @author tags. See the copyright.txt file in the
 * distribution for a full listing of individual contributors.
 *
 * This is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as
 * published by the Free Software Foundation; either version 2.1 of
 * the License, or (at your option) any later version.
 *
 * This software is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this software; if not, write to the Free
 * Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA
 * 02110-1301 USA, or see the FSF site: http://www.fsf.org.
 */

package org.jboss.web.tomcat.security;

import java.io.IOException;
import java.security.Principal;
import java.util.StringTokenizer;

import javax.management.JMException;
import javax.management.ObjectName;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.catalina.Realm;
import org.apache.catalina.Session;
import org.apache.catalina.authenticator.Constants;
import org.apache.catalina.connector.Request;
import org.apache.catalina.connector.Response;
import org.apache.catalina.deploy.LoginConfig;
import org.jboss.logging.Logger;

import org.jboss.as.web.security.ExtendedFormAuthenticator;

/**
 * JBAS-2283: Provide custom header based authentication support
 *
 * Header Authenticator that deals with userid from the request header
Requires
 * two attributes configured on the Tomcat Service - one for the http
header
 * denoting the authenticated identity and the other is the SESSION
cookie
 *
 * @author <a href="mailto:Anil.Saldhana@jboss.org">Anil Saldhana</a>
 * @author <a href="mailto:sguilhen@redhat.com">Stefan Guilhen</a>
 * @version $Revision$
 * @since Sep 11, 2006
 */
public class GenericHeaderAuthenticator extends
ExtendedFormAuthenticator {
  protected static Logger log = Logger
      .getLogger(GenericHeaderAuthenticator.class);

  protected boolean trace = log.isTraceEnabled();

  // JBAS-4804: GenericHeaderAuthenticator injection of ssoid and
  // sessioncookie name.
  private String httpHeaderForSSOAuth = null;
```

```
  private String sessionCookieForSSOAuth = null;

  /**
   * <p>
   * Obtain the value of the <code>httpHeaderForSSOAuth</code>
attribute. This
   * attribute is used to indicate the request header ids that have to
be
   * checked in order to retrieve the SSO identity set by a third party
   * security system.
   * </p>
   *
   * @return a <code>String</code> containing the value of the
   *         <code>httpHeaderForSSOAuth</code> attribute.
   */
  public String getHttpHeaderForSSOAuth() {
    return httpHeaderForSSOAuth;
  }

  /**
   * <p>
   * Set the value of the <code>httpHeaderForSSOAuth</code> attribute.
This
   * attribute is used to indicate the request header ids that have to
be
   * checked in order to retrieve the SSO identity set by a third party
   * security system.
   * </p>
   *
   * @param httpHeaderForSSOAuth
   *            a <code>String</code> containing the value of the
   *            <code>httpHeaderForSSOAuth</code> attribute.
   */
  public void setHttpHeaderForSSOAuth(String httpHeaderForSSOAuth) {
    this.httpHeaderForSSOAuth = httpHeaderForSSOAuth;
  }

  /**
   * <p>
   * Obtain the value of the <code>sessionCookieForSSOAuth</code>
attribute.
   * This attribute is used to indicate the names of the SSO cookies
that may
   * be present in the request object.
   * </p>
   *
   * @return a <code>String</code> containing the names (separated by a
   *         <code>','</code>) of the SSO cookies that may have been
set by a
   *         third party security system in the request.
   */
  public String getSessionCookieForSSOAuth() {
    return sessionCookieForSSOAuth;
  }
```

```
  /**
   * <p>
   * Set the value of the <code>sessionCookieForSSOAuth</code>
attribute. This
   * attribute is used to indicate the names of the SSO cookies that may
be
   * present in the request object.
   * </p>
   *
   * @param sessionCookieForSSOAuth
   *            a <code>String</code> containing the names (separated
by a
   *            <code>','</code>) of the SSO cookies that may have been
set by
   *            a third party security system in the request.
   */
  public void setSessionCookieForSSOAuth(String sessionCookieForSSOAuth)
{
    this.sessionCookieForSSOAuth = sessionCookieForSSOAuth;
  }

  /**
   * <p>
   * Creates an instance of <code>GenericHeaderAuthenticator</code>.
   * </p>
   */
  public GenericHeaderAuthenticator() {
    super();
  }

  public boolean authenticate(Request request, HttpServletResponse
response,
      LoginConfig config) throws IOException {
    log.trace("Authenticating user");

    Principal principal = request.getUserPrincipal();
    if (principal != null) {
      if (trace)
        log.trace("Already authenticated '" + principal.getName() +
"'");
      return true;
    }

    Realm realm = context.getRealm();
    Session session = request.getSessionInternal(true);

    String username = getUserId(request);
    String password = getSessionCookie(request);

    // Check if there is sso id as well as sessionkey
    if (username == null || password == null) {
      log.trace("Username is null or password(sessionkey) is
null:fallback to form auth");
      return super.authenticate(request, response, config);
    }
    principal = realm.authenticate(username, password);
```

```
    if (principal == null) {
      forwardToErrorPage(request, response, config);
      return false;
    }

    session.setNote(Constants.SESS_USERNAME_NOTE, username);
    session.setNote(Constants.SESS_PASSWORD_NOTE, password);
    request.setUserPrincipal(principal);

    register(request, response, principal, HttpServletRequest.FORM_AUTH,
        username, password);
    return true;
  }

  /**
   * Get the username from the request header
   *
   * @param request
   * @return
   */
  protected String getUserId(Request request) {
    String ssoid = null;
    // We can have a comma-separated ids
    String ids = "";
    try {
      ids = this.getIdentityHeaderId();
    } catch (JMException e) {
      if (trace)
        log.trace("getUserId exception", e);
    }
    if (ids == null || ids.length() == 0)
      throw new IllegalStateException(
          "Http headers configuration in tomcat service missing");

    StringTokenizer st = new StringTokenizer(ids, ",");
    while (st.hasMoreTokens()) {
      ssoid = request.getHeader(st.nextToken());
      if (ssoid != null)
        break;
    }
    if (trace)
      log.trace("SSOID-" + ssoid);
    return ssoid;
  }

  /**
   * Obtain the session cookie from the request
   *
   * @param request
   * @return
   */
  protected String getSessionCookie(Request request) {
    Cookie[] cookies = request.getCookies();
    log.trace("Cookies:" + cookies);
    int numCookies = cookies != null ? cookies.length : 0;
```

```
    // We can have comma-separated ids
    String ids = "";
    try {
      ids = this.getSessionCookieId();
      log.trace("Session Cookie Ids=" + ids);
    } catch (JMException e) {
      if (trace)
        log.trace("checkSessionCookie exception", e);
    }
    if (ids == null || ids.length() == 0)
      throw new IllegalStateException(
          "Session cookies configuration in tomcat service missing");

    StringTokenizer st = new StringTokenizer(ids, ",");
    while (st.hasMoreTokens()) {
      String cookieToken = st.nextToken();
      String val = getCookieValue(cookies, numCookies, cookieToken);
      if (val != null)
        return val;
    }
    if (trace)
      log.trace("Session Cookie not found");
    return null;
  }

  /**
   * Get the configured header identity id in the tomcat service
   *
   * @return
   * @throws JMException
   */
  protected String getIdentityHeaderId() throws JMException {
    if (this.httpHeaderForSSOAuth != null)
      return this.httpHeaderForSSOAuth;
    return (String) mserver.getAttribute(new ObjectName(
        "jboss.web:service=WebServer"), "HttpHeaderForSSOAuth");
  }

  /**
   * Get the configured session cookie id in the tomcat service
   *
   * @return
   * @throws JMException
   */
  protected String getSessionCookieId() throws JMException {
    if (this.sessionCookieForSSOAuth != null)
      return this.sessionCookieForSSOAuth;
    return (String) mserver.getAttribute(new ObjectName(
        "jboss.web:service=WebServer"), "SessionCookieForSSOAuth");
  }

  /**
   * Get the value of a cookie if the name matches the token
   *
   * @param cookies
```

```
     *            array of cookies
     * @param numCookies
     *            number of cookies in the array
     * @param token
     *            Key
     * @return value of cookie
     */
    protected String getCookieValue(Cookie[] cookies, int numCookies,
        String token) {
      for (int i = 0; i < numCookies; i++) {
        Cookie cookie = cookies[i];
        log.trace("Matching cookieToken:" + token + " with cookie name="
            + cookie.getName());
        if (token.equals(cookie.getName())) {
          if (trace)
            log.trace("Cookie-" + token + " value=" + cookie.getValue());
          return cookie.getValue();
        }
      }
      return null;
    }
  }
```

バグを報告する

14.3. セキュリティーレルム

14.3.1. セキュリティーレルムについて

セキュリティーレルム はユーザーとパスワード間、およびユーザーとロール間のマッピングです。セキュリティーレルムは EJB や Web アプリケーションに認証や承認を追加するメカニズムです。 JBoss Enterprise Application Platform 6 はデフォルトで次の 2 つのセキュリティーレルムを提供します。

- **ManagementRealm** は、管理 CLI や Web ベースの管理コンソールに機能を提供する管理 API のユーザーやパスワード、ロール情報を保存します。JBoss Enterprise Application Platform を管理するため認証システムを提供します。管理 API に使用する同じビジネスルールでアプリケーションを認証する必要がある場合に **ManagementRealm** を使用することもできます。

- **ApplicationRealm** は Web アプリケーションと EJB のユーザーやパスワード、ロール情報を保存します。

各レルムはファイシステム上の 2 つのファイルに保存されます。

- ***REALM*-users.properties** はユーザー名とハッシュ化されたパスワードを保存します。

- ***REALM*-users.properties** はユーザーからロールへのマッピングを保存します。

プロパティーファイルは **domain/configuration/** および **standalone/configuration/** ディレクトリに保存されます。ファイルは **add-user.sh** や **add-user.bat** コマンドによって同時に書き込まれます。コマンドを実行する時、新しいユーザーをどのレルムに追加するか最初に決定します。

バグを報告する

14.3.2. 新しいセキュリティーレルムの追加

1. **管理 CLI を実行します。**
   jboss-cli.sh または **jboss-cli.bat** コマンドを開始し、サーバーに接続します。

2. **セキュリティーレルムを新規作成します。**
   次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上で **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。

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

3. **新しいロールの情報を保存するプロパティーファイルへの参照を作成します。**
   次のコマンドを実行し、新しいロールに関連するプロパティーが含まれる **myfile.properties** という名前のファイルのポインターを作成します。

注記

新規作成されたプロパティーファイルは、含まれる **add-user.sh** および **add-user.bat** スクリプトによって管理されません。そのため、外部から管理する必要があります。

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

結果

セキュリティーレルムが新規作成されます。この新規作成されたレルムにユーザーやロールを追加すると、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情報が保存されます。新規ファイルはご使用のアプリケーションやプロシージャーを使用して管理できます。

バグを報告する

14.3.3. セキュリティーレルムへユーザーを追加

1. **add-user.sh または add-user.bat コマンドを実行します。**
   コマンドラインインターフェース (CLI) を開きます。***EAP_HOME*/bin/** ディレクトリへ移動します。Red Hat Enterprise Linux や他の UNIX 系のオペレーティングシステムを稼働している場合は **add-user.sh** を実行します。Microsoft Windows Server を稼働している場合は **add-user.bat** を実行します。

2. **管理ユーザーかアプリケーションユーザーのどちらを追加するか選択します。**
   この手順では **b** を入力し、アプリケーションユーザーを追加します。

3. **ユーザーが追加されるレルムを選択します。**
   デフォルトでは、**ApplicationRealm** のみが選択可能です。カスタムレルムが追加されている場合はその名前を入力します。

4. **入力を促されたらユーザー名、パスワード、ロールを入力します。**
   入力を促されたら希望のユーザー名、パスワード、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** を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。

バグを報告する

14.4. EJB アプリケーションセキュリティー

14.4.1. セキュリティアイデンティティ (ID)

14.4.1.1. EJB のセキュリティーアイデンティティーについて

*セキュリティーアイデンティティー*は *呼び出しアイデンティティー*とも呼ばれており、セキュリティー設定では **<security-identity>** タグのことです。これは、EJB がコンポーネントでメソッド呼び出しを行う際に必ず使う必要のあるアイデンティティーを指します。

呼び出しアイデンティティーは現在の呼び出し元か特定のロールのいずれかになります。現在の呼び出し元である場合は、**<use-caller-identity>** タグがあり、特定ロールの場合は**<run-as>** タグが使用されます。

EJB のセキュリティーアイデンティティー設定に関する情報は 「EJB のセキュリティーアイデンティティーの設定」 を参照してください。

バグを報告する

14.4.1.2. EJB のセキュリティーアイデンティティーの設定

例14.4 呼び出し側と同じになるように EJB のセキュリティーアイデンティティーを設定する

この例は、現在の呼び出し側のアイデンティティーと同じになるように、EJB によって実行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設定します。**<security-identity>** 要素の宣言を指定しない場合、この挙動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14.5 特定ロールに EJB のセキュリティーアイデンティティーを設定する

特定のロールにセキュリティーアイデンティティーを設定するには、**<security-identity>** タグの中に **<run-as>** または **<role>** 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へ割り当てられます。違うプリンシパルを割り当てる場合は **<run-as-principle>** 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

サーブレット要素内に **<run-as>** 要素と **<run-as-principal>** 要素を使用することもできます。

以下も参照してください。

- 「EJB のセキュリティーアイデンティティーについて」
- 「EJB セキュリティーパラメーターについての参考資料」

バグを報告する

14.4.2. EJB メソッドのパーミッション

14.4.2.1. EJB メソッドパーミッションについて

EJB は **<method-permisison>** 要素の宣言を提供します。この宣言により、EJB のインターフェースメソッドを呼び出し可能なロールを設定します。以下の組み合わせに対してパーミッションの指定が可能です。

- 名前付き EJB のホームおよびコンポーネントインターフェースメソッド
- 名前付き EJB のホームあるいはコンポーネントインターフェースの指定メソッド
- オーバーロードした名前を持つメソッドセット内の指定メソッド

例は 「EJB メソッドパーミッションの使用」 を参照してください。

バグを報告する

14.4.2.2. EJB メソッドパーミッションの使用

概要

<method-permission> 要素は、**<method>** 要素によって定義される EJB メソッドへアクセスできる論理ロールを定義します。XML の構文を表す例は複数あります。メソッドパーミッションステートメントは複数存在することがあり、累積的な影響があります。**<method-permission>** 要素は **<ejb-jar>** 記述子の **<assembly-descriptor>** 要素の子要素です。

XML 構文は、EJB メソッドへセキュリティーアノテーションを使用することの代替となります。

例14.6 ロールが EJB の全メソッドへのアクセスできるようにする

```
<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14.7 EJB の特定メソッドへのみロールがアクセスできるようにし、パラメーターを渡すことができるメソッドを制限する

```
<method-permission>
  <description>The employee role may access the findByPrimaryKey,
  getEmployeeInfo, and the updateEmployeeInfo(String) method of
  the AcmekPayroll bean </description>
  <role-name>employee</role-name>
  <method>
 <ejb-name>AcmekPayroll</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14.8 認証された全ユーザーが EJB のメソッドにアクセスできるようにする

<unchecked/> 要素を使用すると、認証された全ユーザーが指定の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14.9 特定の EJB メソッドを完全に除外して使用されないようにする

```
<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14.10 複数の <method-permission> ブロックが含まれる完全な <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>
```

バグを報告する

14.4.3. EJB セキュリティアノテーション

14.4.3.1. EJB セキュリティーアノテーションについて

EJB はセキュリティーアノテーションを使いセキュリティー関連の情報をデプロイヤーに渡します。セキュリティーアノテーションには以下が含まれます。

@DeclareRoles

どのロールが利用可能か宣言します。

@SecurityDomain

EJB に使用するセキュリティードメインを指定します。承認のため、EJB に **@RolesAllowed** アノテーションが付いている場合、EJB にセキュリティードメインがアノテーション付けされている場合のみ承認を適用します。

@RolesAllowed、@PermitAll、@DenyAll

どのメソッドパーミッションが可能か指定します。メソッドパーミッションについては 「EJB メソッドパーミッションについて」 を参照してください。

@RolesAllowed、@PermitAll、@DenyAll

どのメソッドパーミッションが可能か指定します。メソッドパーミッションについては 「EJB メソッドパーミッションについて」 を参照してください。

@RunAs

コンポーネントの伝搬されたセキュリティー ID を設定します。

詳細は 「EJB セキュリティーアノテーションの使用」 を参照してください。

バグを報告する

14.4.3.2. EJB セキュリティーアノテーションの使用

概要

XML 記述子かアノテーションを使用して、どのセキュリティーロールが Enterprise JavaBean (EJB) でメソッドを呼び出しできるかを制御することができます。XML 記述子の使用については 「EJB メソッドパーミッションの使用」 を参照してください。

EJB 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、どのセキュリティーロールに対してパーミッションをチェックするか定義します。@DeclareRoles が存在しない場合、@RolesAllowed アノテーションよりリストが自動的に構築されます。

@SecurityDomain

EJB に使用するセキュリティードメインを指定します。承認のため、EJB に **@RolesAllowed** アノテーションが付いている場合、EJB にセキュリティードメインがアノテーション付けされている場合のみ承認を適用します。

@RolesAllowed、@PermitAll、@DenyAll

@RolesAllowed を使用して、1 つまたは複数のメソッドへのアクセスが許可されるロールをリストします。すべてのロールに対して 1 つまたは複数のメソッドの使用を許可する場合は @PermitAll、すべてのロールに対してメソッドの使用を拒否する場合は @DenyAll を使用します。

@RunAs

@RunAs を使用してロールを指定すると、メソッドが常にそのロールとして実行されるようにします。

例14.11 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
 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 **tempemployee** ロールとして実行されます。**GoodbyeAdmin** メソッドと、セキュリティーアノテーションのない他のメソッドへは **admin** ロールのみがアクセスできます。

バグを報告する

14.4.4. EJB へのリモートアクセス

14.4.4.1. リモートメソッドアクセスについて

JBoss Remoting は EJB や JMX MBeans、その他類似のサービスにリモートアクセスを提供するフレームワークです。SSL の有無にかかわらず次のトランスポートタイプ内で動作します。

サポートされているトランスポートタイプ

- ソケット / セキュアソケット
- RMI / RMI over SSL
- HTTP / HTTPS
- サーブレット / セキュアサーブレット
- バイソケット (Bisocket) / セキュアバイソケット (Secure Bisocket)

JBoss Remoting はマルチキャストまたは JNDI からの自動ディスカバリーも提供します。

自動ディスカバリーは JBoss Enterprise Application Platform 内の多くのサブシステムによって使用さ

れています。これにより、複数の異なるトランスポートメカニズム上でクライアントによってリモートで呼び出されるサービスを設計、実装、デプロイすることが可能になります。さらに、JBoss Enterprise Application Platform の既存サービスへのアクセスが可能になります。

データのマーシャリング

Remoting システムはデータのマーシャリングサービスやアンマーシャリングサービスも提供します。データのマーシャリングとは、別のシステムで処理を実行できるようネットワークやプラットフォーム境界の全体で安全にデータを移動できる機能のことを言います。処理結果は元のシステムへ返送され、ローカルで処理されたように動作します。

アーキテクチャーの概要

Remoting を使用するクライアントアプリケーションを設計する場合、URL 型の形式の単純な文字列である **`InvokerLocator`** と呼ばれる特別なリソースロケーターを使用するよう設定し、アプリケーションがサーバーと通信するようにします。**`remoting`** サブシステムの一部として設定される**`connector`** 上で、サーバーはリモートリソースの要求をリッスンします。**`connector`** は設定済みの **`ServerInvocationHandler`** へ要求を渡します。各 **`ServerInvocationHandler`** は要求の対処方法を認識するメソッド **`invoke(InvocationRequest)`** を実装します。

JBoss Remoting フレームワークにはクライアントとサーバー側でお互いをミラーリングする 3 つのレイヤーが含まれています。

JBoss Remoting フレームワークレイヤー

- ユーザーは外部レイヤーとやりとりします。クライアント側では外部レイヤーは呼び出し要求を送信する **`Client`** クラスになります。サーバー側ではユーザーによって実装され、呼び出し要求を受信する InvocationHandler になります。
- トランスポートはインボーカーレイヤーによって制御されます。
- 最も下のレイヤーにはデータ形式をワイヤー形式に変換するマーシャラーとアンマーシャラーが含まれています。

バグを報告する

14.4.4.2. Remoting コールバックについて

Remoting クライアントがサーバーからの情報を要求する時、サーバーをブロックし、サーバーの返答を待つことが可能ですが、この挙動は多くの場合で理想的ではありません。クライアントがサーバー上で非同期イベントをリッスンできるようにし、サーバーが要求の処理を終了するまでクライアントが別の作業を継続できるようにするには、サーバーが要求の処理を終了した時に通知を送信するようアプリケーションが要求するようにします。これをコールバックと呼びます。他のクライアントの代わりに生成された非同期イベントに対してクライアントは自身をリスナーとして追加することもできます。コールバックの受信方法には、プルコールバックとプッシュコールバックの 2 つの方法があります。クライアントはプルコールバックを同期的に確認しますが、プッシュコールバックは受動的にリッスンします。

基本的に、コールバックではサーバーが **`InvocationRequest`** をクライアントに送信します。コールバックが同期的または非同期的であるかに関わらず、サーバー側のコードは同様に動作します。クライアントのみが違いを認識する必要があります。サーバーの InvocationRequest は **`responseObject`** をクライアントに送信します。これはクライアントが要求したペイロードで、要求やイベント通知への直接応答になる場合があります。

また、サーバーは **`m_listeners`** オブジェクトを使用してリスナーを追跡します。これにはサーバーハンドラーに追加された全リスナーのリストが含まれます。**`ServerInvocationHandler`** インターフェースにはこのリストを管理できるようにするメソッドが含まれます。

クライアントはプルコールバックとプッシュコールバックを異なる方法で対応します。どちらの場合でもコールバックハンドラーを実装する必要があります。コールバックハンドラーはインターフェース **`org.jboss.remoting.InvokerCallbackHandler`** の実装で、コールバックデータを処理します。コールバックハンドラーの実装後、プルコールバックのリスナーを追加するか、プッシュコールバックのコールバックサーバーを実装します。

プルコールバック

プルコールバックでは、**`Client.addListener()`** メソッドを使用してクライアントが自身にサーバーのリスナーリストを追加します。その後、コールバックデータを同期的に配信するためサーバーを周期的にプルします。ここでは **`Client.getCallbacks()`** を使用してプルが実行されます。

プッシュコールバック

プッシュコールバックではクライアントアプリケーションが独自の InvocationHandler を実行する必要があります。これには、クライアント上で Remoting サービスを実行する必要があります。これは *コールバックサーバー*と呼ばれます。コールバックサーバーは受信する要求を非同期的に許可し、要求元 (この場合はサーバー) のために処理します。メインサーバーを用いてクライアントのコールバックサーバーを登録するには、コールバックサーバーの **`InvokerLocator`** を **`addListener`** への 2 番目の引数として渡します。

バグを報告する

14.4.4.3. リモーティングサーバーの検出について

リモーティングサーバーとクライアントは JNDI またはマルチキャストを使用してお互いを自動的に検出することができます。リモーティングディテクターはクライアントとサーバーの両方に追加され、NetworkRegistry はクライアントに追加されています。

サーバー側のディテクターは InvokerRegistry を周期的にスキャンし、作成したサーバーインボーカーをすべてプルします。この情報を使用して、ロケーターや各サーバーインボーカーによってサポートされるサブシステムが含まれる検出メッセージを公開します。マルチキャストブロードキャストよりメッセージを公開するか、JNDI サーバーへバインドしてメッセージを公開します。

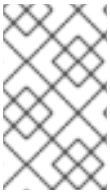
クライアント側ではディテクターはマルチキャストメッセージを受信したり、JNDI サーバーを周期的にポーリングして検出メッセージを読み出します。検出メッセージが新たに検出されたリモーティングサーバーに対するメッセージであることが分かると、ディテクターは NetworkRegistry へ登録します。また、ディテクターは使用できないサーバーを検出すると、NetworkRegistry を更新します。

バグを報告する

14.4.4.4. Remoting サブシステムの設定

概要

JBoss Remoting にはワーカースレッドプール、1 つ以上のコネクター、複数のローカルおよびリモート接続 URI の 3 つのトップレベル設定可能要素があります。ここでは設定可能な項目の説明、各項目の設定方法に対する CLI コマンド例、完全設定されたサブシステムの XML 例について取り上げます。この設定はサーバーのみに適用されます。独自のアプリケーションにカスタムコネクターを使用する場合を除き、Remoting のサブシステムの設定は必要でないことがほとんどです。EJB など Remoting クライアントとして動作するアプリケーションには特定のコネクターに接続するための個別の設定が必要になります。



注記

Remoting サブシステムの設定は Web ベースの管理コンソールには公開されませんが、コマンドラインベースの管理 CLI より完全に設定することが可能です。手作業で XML を編集することは推奨されません。

CLI コマンドの適合

デフォルトの **default** プロファイルを設定する時の CLI コマンドについて説明します。異なるプロファイルを設定するには、プロファイルの名前を置き換えます。スタンドアロンサーバーではコマンドの **/profile=default** の部分を省略します。

Remoting サブシステム外部の設定

remoting サブシステム外部となる設定もあります。

ネットワークインターフェース

remoting サブシステムによって使用されるネットワークネットワークインターフェースは **domain/configuration/domain.xml** または **standalone/configuration/standalone.xml** で定義される **unsecure** インターフェースです。

```
<interfaces>
   <interface name="management"/>
   <interface name="public"/>
   <interface name="unsecure"/>
</interfaces>
```

unsecure インターフェースのホストごとの定義は**domain.xml** または **standalone.xml** と同じディレクトリにある **host.xml** で定義されます。また、このインターフェースは複数の他のサブシステムによっても使用されます。変更する場合は十分注意してください。

```
<interfaces>
   <interface name="management">
      <inet-address
value="${jboss.bind.address.management:127.0.0.1}"/>
   </interface>
   <interface name="public">
      <inet-address value="${jboss.bind.address:127.0.0.1}"/>
   </interface>
   <interface name="unsecure">
      <!-- Used for IIOP sockets in the standard configuration.
         To secure JacORB you need to setup SSL -->
      <inet-address value="${jboss.bind.address.unsecure:127.0.0.1}"/>
   </interface>
</interfaces>
```

socket-binding

remoting サブシステムによって使用されるデフォルトの socket-binding は TCP ポート **4777** へバインドします。この設定を変更する必要がある場合はソケットバインディングとソケットバインディンググループに関するドキュメントを参照してください。

EJB のリモーティングコネクター参照

EJB サブシステムにはリモートメソッド呼び出しに対するリモーティングコネクターへの参照が含まれています。デフォルト設定は次の通りです。

```
<remote connector-ref="remoting-connector" thread-pool-name="default"/>
```

セキュアなトランスポート設定

Remoting はクライアントの要求があれば StartTLS を使用してセキュアな接続 (HTTPS、Secure Servlet など) を使用します。セキュアな接続とセキュアでない接続の両方で同じソケットバインディング (ネットワークポート) が使用されるため、サーバー側に追加の設定をする必要はありません。クライアントはニーズに従ってセキュアなトランスポートまたはセキュアでないトランスポートを要求します。EJB、ORB、JMS プロバイダーなどの Remoting を使用する JBoss Enterprise Application Platform のコンポーネントはデフォルトでセキュアなインターフェースを使用します。

警告

StartTLS はクライアントの要求があればセキュアな接続を有効にしますが、セキュアでない接続がデフォルトになります。本質的に、StartTLS は攻撃者がクライアントの要求を妨害し、要求を編集してセキュアでない接続を要求する*中間者攻撃*の対象になりやすい欠点があります。セキュアでない接続が適切なフォールバックである場合を除き、クライアントがセキュアな接続を取得できなかったときに適切に失敗するよう記述する必要があります。

ワーカースレッドプール

ワーカースレッドプールは、Remoting コネクターからの作業を処理できるスレッドのグループのことです。単一の要素 **<worker-thread-pool>** で、複数の属性を取ります。ネットワークタイムアウトやスレッド不足が発生したり、メモリーの使用を制限する場合にこれらの属性を調節します。特定の推奨設定は状況によって異なります。詳細は Red Hat グローバルサポートサービスまでご連絡ください。

表14.2 ワーカースレッドプールの属性

属性	説明	CLI コマンド
read-threads	リモーティングワーカーに対して作成する読み取りスレッドの数。デフォルトは **1** です。	**/profile=default/subsystem=remoting/:write-attribute(name=worker-read-threads,value=1)**

属性	説明	CLI コマンド
write-threads	リモーティングワーカーに対して作成する書き込みスレッドの数。デフォルトは **1** です。	**/profile=default/subsystem=remoting/:write-attribute(name=worker-write-threads,value=1)**
task-keepalive	コアでないリモーティングワーカーのタスクスレッドを生存させておく期間 (ミリ秒単位) です。デフォルトは **60** です。	**/profile=default/subsystem=remoting/:write-attribute(name=worker-task-keepalive,value=60)**
task-max-threads	モーティングワーカーのタスクスレッドプールに対するスレッドの最大数です。デフォルトは **16** です。	**/profile=default/subsystem=remoting/:write-attribute(name=worker-task-max-threads,value=16)**
task-core-threads	モーティングワーカーのタスクスレッドプールに対するコアスレッドの数です。デフォルトは **4** です。	**/profile=default/subsystem=remoting/:write-attribute(name=worker-task-core-threads,value=4)**
task-limit	挿入前に許可されるリモーティングワーカータスクの最大数です。デフォルトは **16384** です。	**/profile=default/subsystem=remoting/:write-attribute(name=worker-task-limit,value=16384)**

コネクター

コネクターは主な Remoting 設定要素です。複数のコネクターを設定できます。各コネクターは、サブ要素を持つ **<connector>** 要素より構成され、複数の属性が含まれることもあります。デフォルトのコネクターは JBoss Enterprise Application Platform の複数のサブシステムによって使用されます。カスタムコネクターの要素や属性の設定はアプリケーションによって異なるため、詳細は Red Hat グローバルサポートサービスまでご連絡ください。

表14.3 コネクターの属性

属性	説明	CLI コマンド
name	JNDI によって使用されるコネクターの名前です。	**/profile=default/subsystem=remoting/connector=remoting-connector/:write-attribute(name=name,value=remoting-connector)**

属性	説明	CLI コマンド
socket-binding	このコネクターに使用するソケットバインディングの名前です。	`/profile=default/subsystem=remoting/connector=remoting-connector/:write-attribute(name=socket-binding,value=remoting)`
authentication-provider	このコネクターと使用する JASPIC (Java Authentication Service Provider Interface) モジュールです。このモジュールはクラスパスに存在しなければなりません。	`/profile=default/subsystem=remoting/connector=remoting-connector/:write-attribute(name=authentication-provider,value=myProvider)`
security-realm	任意の設定です。アプリケーションのユーザーやパスワード、ロールが含まれるセキュリティーレルムになります。EJB または Web アプリケーションがセキュリティーレルムに対して認証を行います。 **ApplicationRealm** はデフォルトの JBoss Enterprise Application Platform インストールで使用可能です。	`/profile=default/subsystem=remoting/connector=remoting-connector/:write-attribute(name=security-realm,value=ApplicationRealm)`

表14.4 コネクター要素

属性	説明	CLI コマンド
sasl	SASL (Simple Authentication and Security Layer) 認証メカニズムの囲み要素です。	**N/A**
properties	1 つ以上の **<property>** 要素が含まれ、各要素には **name** 属性と任意の **value** 属性が含まれます。	`/profile=default/subsystem=remoting/connector=remoting-connector/property=myProp/:add(value=myPropValue)`

送信接続

3 つのタイプの送信接続を指定することができます。

- URI への送信接続
- ローカルの送信接続 – ソケットなどのローカルリソースへ接続します。
- リモートの送信接続 – リモートリソースへ接続し、セキュリティーレルムを使用して認証を行います。

送信接続はすべて **<outbound-connections>** 要素で囲まれます。各接続タイプは **outbound-socket-binding-ref** 属性を取ります。送信接続は **uri** 属性を取ります。リモートの送信接続は任意の **username** 属性と **security-realm** 属性を取り、認証に使用します。

表14.5 送信接続要素

属性	説明	CLI コマンド
outbound-connection	汎用の送信接続。	**/profile=default/subsystem=remoting/outbound-connection=my-connection/:add(uri=http://my-connection)**
local-outbound-connection	暗黙の local:// URI スキームを持つ送信接続。	**/profile=default/subsystem=remoting/local-outbound-connection=my-connection/:add(outbound-socket-binding-ref=remoting2)**
remote-outbound-connection	セキュリティーレルムを用いた基本またはダイジェスト認証を使用する remote:// URI スキームの送信接続です。	**/profile=default/subsystem=remoting/remote-outbound-connection=my-connection/:add(outbound-socket-binding-ref=remoting,username=myUser,security-realm=ApplicationRealm)**

SASL 要素

SASL 子要素を定義する前に初期 SASL 要素を作成する必要があります。次のコマンドを使用します。

```
/profile=default/subsystem=remoting/connector=remoting-
connector/security=sasl:add
```

SASL 要素の子要素は次の表の通りです。

属性	説明	CLI コマンド
include-mechanisms	SASL メカニズムのスペース区切りのリストである **value** 属性が含まれています。	`/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl:write-attribute(name=include-mechanisms,value=["DIGEST","PLAIN","GSSAPI"])`

属性	説明	CLI コマンド
qop	SASL の保護品質値が希望順に並ぶスペース区切りのリストである **value** 属性が含まれます。	`/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl:write-attribute(name=qop,value=["auth"])`
strength	SASL の暗号強度の値が希望順に並ぶスペース区切りのリストである **value** 属性が含まれます。	`/subsystem=remoting/connector=remoting-connector/security=sasl:write-attribute(name=strength,value=["medium"])`
reuse-session	ブール値である **value** 属性が含まれます。**true** の場合、セッションの再使用を試みます。	`/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl:write-attribute(name=reuse-session,value=false)`
server-auth	ブール値である **value** 属性が含まれます。**true** の場合、サーバーはクライアントに対して認証します。	`/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl:write-attribute(name=server-auth,value=false)`
policy	以下の要素がゼロ個以上含まれ、各要素が単一の **value** を取るエンクロージング要素です。 • forward-secrecy – メカニズムによるフォワード秘匿性 (forward secrecy) の実装が必要であるかどうか (あるセッションが侵入されても、その後のセッションへの侵入に関する情報は自動的に提供されません)。	`/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/sasl-policy=policy:add` `/profile=default/subsystem=remoting/connector=remoting-`

属性	説明	CLI コマンド
	• no-active – 辞書攻撃でない攻撃を受けやすいメカニズムを許可するかどうか。値が **false** の場合は許可し、 **true** の場合は許可しません。 • no-anonymous – 匿名ログインを許可するメカニズムを許可するかどうか。値が **false** の場合は許可し、 **true** の場合は許可しません。 • no-dictionary – 受動的な辞書攻撃を受けやすいメカニズムを許可するかどうか。値が **false** の場合は許可し、 **true** の場合は許可しません。 • no-plain-text – 単純で受動的な辞書攻撃を受けやすいメカニズムを許可するかどうか。値が **false** の場合は許可し、 **true** の場合は許可しません。 • pass-credentials – クライアントの認証情報を渡すメカニズムを許可するかどうか。	connector/security=sasl/sasl-policy=policy:write-attribute(name=forward-secrecy,value=true) /profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/sasl-policy=policy:write-attribute(name=no-active,value=false) /profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/sasl-policy=policy:write-attribute(name=no-dictionary,value=true) /profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/sasl-policy=policy:write-attribute(name=no-plain-text,value=false) /profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/sasl-policy=policy:write-attribute(name=pass-credentials,value=true)

属性	説明	CLI コマンド
properties	1 つ以上の **<property>** 要素が含まれ、各要素には **name** 属性と任意の **value** 属性が含まれます。	`/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/property=myprop:add(value=1)` `/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/property=myprop2:add(value=2)`

例14.12 設定例

この例は JBoss Enterprise Application Platform 6 のデフォルトのリモーティングサブシステムを表しています。

```
<subsystem xmlns="urn:jboss:domain:remoting:1.1">
    <connector name="remoting-connector" socket-binding="remoting"
security-realm="ApplicationRealm"/>
</subsystem>
```

この例には多くの仮説的な値が含まれており、前述の要素や属性がコンテキストに含まれています。

```
<subsystem xmlns="urn:jboss:domain:remoting:1.1">
    <worker-thread-pool read-threads="1" task-keepalive="60' task-max-
threads="16" task-core-thread="4" task-limit="16384" write-threads="1"
/>
    <connector name="remoting-connector" socket-binding="remoting"
security-realm="ApplicationRealm">
        <sasl>
            <include-mechanisms value="GSSAPI PLAIN DIGEST-MD5" />
            <qop value="auth" />
            <strength value="medium" />
            <reuse-session value="false" />
            <server-auth value="false" />
            <policy>
                <forward-secrecy value="true" />
                <no-active value="false" />
                <no-anonymous value="false" />
                <no-dictionary value="true" />
                <no-plain-text value="false" />
                <pass-credentials value="true" />
            </policy>
```

```
            <properties>
                <property name="myprop1" value="1" />
                <property name="myprop2" value="2" />
            </properties>
        </sasl>
        <authentication-provider name="myprovider" />
        <properties>
            <property name="myprop3" value="propValue" />
        </properties>
    </connector>
    <outbound-connections>
        <outbound-connection name="my-outbound-connection" uri="htt\
p://myhost:7777/"/>
        <remote-outbound-connection name="my-remote-connection"
outbound-socket-binding-ref="my-remote-socket" username="myUser"
security-realm="ApplicationRealm"/>
        <local-outbound-connection name="myLocalConnection" outbound-
socket-binding-ref="my-outbound-socket"/>
    </outbound-connections>
</subsystem>
```

文書化されていない設定の側面

- JIDI および マルチキャスト自動検出

バグを報告する

14.4.4.5. リモート EJB クライアントを用いたセキュリティーレルムの使用

セキュリティーレルムの使用は、リモートで EJB を呼び出すクライアントへセキュリティーを追加する 1 つの方法です。セキュリティーレルムはユーザー名とパスワードのペアとユーザー名とロールのペアの単純なデータベースです。セキュリティーレノムという言葉は Web コンテナに関しても使用されますが、若干意味が異なります。

次の手順に従って、セキュリティーレルムに存在する特定のユーザー名やパスワードに対して EJB を認証します。

- 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かスタンドアロンサーバーに追加します。
- 次のパラメーターをアプリケーションのクラスパスにある **jboss-ejb-client.properties** ファイルに追加します。この例では、ファイルの他のパラメーターは接続を **default** として見なすことを前提とします。

  ```
  ¶
  remote.connection.default.username=appuser¶
  remote.connection.default.password=apppassword¶
  ```

- 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上にカスタム Remoting コネクターを作成します。

- カスタム Remoting コネクターを用いてプロファイルを使用するよう設定されているサーバーグループに EJB をデプロイします。管理された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。

バグを報告する

14.4.4.6. 新しいセキュリティーレルムの追加

1. **管理 CLI を実行します。**
   jboss-cli.sh または **jboss-cli.bat** コマンドを開始し、サーバーに接続します。

2. **セキュリティーレルムを新規作成します。**
   次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上で **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。

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

3. **新しいロールの情報を保存するプロパティーファイルへの参照を作成します。**
   次のコマンドを実行し、新しいロールに関連するプロパティーが含まれる **myfile.properties** という名前のファイルのポインターを作成します。

注記

新規作成されたプロパティーファイルは、含まれる **add-user.sh** および **add-user.bat** スクリプトによって管理されません。そのため、外部から管理する必要があります。

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

結果

セキュリティーレルムが新規作成されます。この新規作成されたレルムにユーザーやロールを追加すると、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情報が保存されます。新規ファイルはご使用のアプリケーションやプロシージャーを使用して管理できます。

バグを報告する

14.4.4.7. セキュリティーレルムへユーザーを追加

1. **add-user.sh または add-user.bat コマンドを実行します。**
   コマンドラインインターフェース (CLI) を開きます。***EAP_HOME*/bin/** ディレクトリへ移動します。Red Hat Enterprise Linux や他の UNIX 系のオペレーティングシステムを稼働している場合は **add-user.sh** を実行します。Microsoft Windows Server を稼働している場合は **add-user.bat** を実行します。

2. **管理ユーザーかアプリケーションユーザーのどちらを追加するか選択します。**
   この手順では **b** を入力し、アプリケーションユーザーを追加します。

3. **ユーザーが追加されるレルムを選択します。**

デフォルトでは、**ApplicationRealm** のみが選択可能です。カスタムレルムが追加されている場合はその名前を入力します。

4. **入力を促されたらユーザー名、パスワード、ロールを入力します。**
入力を促されたら希望のユーザー名、パスワード、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** を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。

バグを報告する

14.4.4.8. SSL による暗号化を使用したリモート EJB アクセスについて

デフォルトでは、EJB2 および EJB3 Bean の RMI (リモートメソッド呼び出し) に対するネットワークトラフィックは暗号化されていません。暗号化が必要な場合、SSL (セキュアソケットレイヤー) を使いクライアントとサーバー間の接続が暗号化されるようにします。SSL には、 RMI ポートをブロックするファイアウォールをネットワークトラフィックが横断できる利点があります。

バグを報告する

14.5. JAX-RS アプリケーションセキュリティー

14.5.1. RESTEasy JAX-RS Web サービスのロールベースのセキュリティーを有効にする

概要

RESTEasy は JAX-RS メソッドの @RolesAllowed、@PermitAll、@DenyAll アノテーションをサポートしますが、デフォルトではこれらのアノテーションを認識しません。次の手順に従って **web.xml** ファイルを設定し、ロールベースセキュリティーを有効にします。

警告

アプリケーションが EJB を使用する場合はロールベースセキュリティーを有効にしないでください。RESTEasy ではなく EJB コンテナが機能を提供します。

手順14.3 RESTEasy JAX-RS Web サービスのロールベースのセキュリティーを有効にする

1. テキストエディターでアプリケーションの **web.xml** ファイルを開きます。
2. 以下の **<context-param>** をファイルの **web-app** タグ内に追加します。

```
<context-param>
    <param-name>resteasy.role.based.security</param-name>
    <param-value>true</param-value>
</context-param>
```

3. **<security-role>** タグを使用して RESTEasy JAX-RS WAR ファイル内で使用されるすべてのロールを宣言します。

```
<security-role><role-name>ROLE_NAME</role-name></security-role>
<security-role><role-name>ROLE_NAME</role-name></security-role>
```

4. すべてのロールに対して JAX-RS ランタイムが対応する全 URL へのアクセスを承認します。

```
<security-constraint><web-resource-collection><web-resource-
name>Resteasy</web-resource-name><url-pattern>/PATH</url-pattern>
</web-resource-collection><auth-constraint><role-
name>ROLE_NAME</role-name><role-name>ROLE_NAME</role-name></auth-
constraint></security-constraint>
```

結果

ロールベースセキュリティーが定義されたロールのセットによりアプリケーション内で有効になります。

例14.13 ロールベースセキュリティーの設定例

```
<web-app>

   <context-param>
 <param-name>resteasy.role.based.security</param-name>
 <param-value>true</param-value>
   </context-param>

   <servlet-mapping>
 <servlet-name>Resteasy</servlet-name>
 <url-pattern>/*</url-pattern>
   </servlet-mapping>

   <security-constraint>
 <web-resource-collection>
     <web-resource-name>Resteasy</web-resource-name>
     <url-pattern>/security</url-pattern>
 </web-resource-collection>
 <auth-constraint>
     <role-name>admin</role-name>
     <role-name>user</role-name>
 </auth-constraint>
   </security-constraint>

   <security-role>
 <role-name>admin</role-name>
```

```
    </security-role>
    <security-role>
  <role-name>user</role-name>
    </security-role>

</web-app>
```

バグを報告する

14.5.2. アノテーションを使用した JAX-RS Web サービスの保護

概要

サポート対象のセキュリティーアノテーションを使用して JAX-RS Web サービスを保護する手順を取り上げます。

手順14.4 サポート対象のセキュリティーアノテーションを使用した JAX-RS Web サービスのセキュア化

1. ロールベースセキュリティーを有効にします。詳細は 「RESTEasy JAX-RS Web サービスのロールベースのセキュリティーを有効にする」 を参照してください。
2. JAX-RS Web サービスにセキュリティーアノテーションを追加します。RESTEasy は次のアノテーションをサポートします。

   @RolesAllowed

   メソッドにアクセスできるロールを定義します。ロールはすべて `web.xml` ファイルに定義する必要があります。

   @PermitAll

   `web.xml` ファイルに定義されている全ロールによるメソッドへのアクセスを許可します。

   @DenyAll

   メソッドへのアクセスをすべて拒否します。

バグを報告する

14.6. リモートパスワードプロトコルの保護

14.6.1. SRP (セキュアリモートパスワード) プロトコルについて

SRP (セキュアリモートパスワード) プロトコルは、Internet Standards Working Group の Request For Comments 2945 (RFC2945) に記載されている、公開鍵交換のハンドシェイク実装です。RFC2945 の要約は次のようになります。

> この文書は SRP (セキュアリモートパスワード) プロトコルとして知られる、強固なネットワーク暗号化方式について説明しています。このメカニズムは従来の再利用可能なパスワードで起きていたセキュリティーの問題を排除しつつ、ユーザーによって提供されるパスワードを使いセキュアな接続をネゴシエーションするのに適しています。この仕組みは、認証プロセスでセキュアな鍵交換を行い、セッション時にセキュ

リティー層 (プライバシーや整合性保護) を有効にすることが可能です。信頼されたキーサーバーや証明書インフラストラクチャーは必要なく、また長期鍵の保存や管理にクライアントを必要としません。SRP には、既存のチャレンジレスポンス方式と比べセキュリティーやデプロイメントに両方において様々な利点があり、SRP は安全なパスワード認証が必要な場合に、互換性のある理想的な代用方式となります。

完全な RFC2945 仕様は http://www.rfc-editor.org/rfc.html から取得可能です。SRP アルゴリズムに関する追加情報および履歴については http://www.rfc-editor.org/rfc.html を参照してください。

Diffie-Hellman および RSA などのアルゴリズムは公開鍵交換アルゴリズムとして知られています。公開鍵アルゴリズムのコンセプトには、誰でも使用できる公開鍵と本人のみが把握する秘密鍵の 2 つの鍵が存在します。暗号化した情報を送信したい場合、この公開鍵を使い情報を暗号化します。秘密鍵を持つ本人のみが秘密鍵を使い情報を復号化することができます。従来の共有パスワードベースの暗号化方式では、受信者も送信者も共有パスワードを把握している必要があります。公開鍵アルゴリズムではパスワードを共有する必要がありません。

バグを報告する

14.6.2. セキュアリモートパスワード (SRP) プロトコルの設定

セキュアリモートパスワード (SRP) プロトコルをアプリケーションで使用するには、最初に **SRPVerifierStore** インターフェースを実装する MBean を作成します。実装に関する詳細は SRPVerifierStore 実装 で確認できます。

手順14.5 既存パスワードストアの統合

1. **ハッシュ化されたパスワード情報ストアを作成します。**
   パスワードが既に不可逆的にハッシュ化され、保存されている場合、この作業をユーザーごとに行う必要があります。

   noOP メソッドとして **`setUserVerifier(String, VerifierInfo)`** を実装するか、ストアが読み取り専用であることを知らせる例外をスローするメソッドとして **`setUserVerifier(String, VerifierInfo)`** を実装することができます。

2. **SRPVerifierStore インターフェースを作成します。**
   作成したストアより **`VerifierInfo`** を取得できるカスタムの **`SRPVerifierStore`** インターフェース実装を作成します。

   `verifyUserChallenge(String, Object)` を使用すると、SafeWord や Radius のような既存のハードウェアトークンベースのスキームを SRP アルゴリズムへ統合することができます。このインターフェースメソッドは、クライアントの SRPLoginModule 設定で hasAuxChallenge オプションが指定されている場合のみ呼び出されます。

3. **JNDI MBean を作成します。**
   JNDI が使用できる **`SRPVerifierStore`** インターフェースを公開し、必要な設定可能パラメーターを公開する MBean を作成します。

   デフォルトの **`org.jboss.security.srp.SRPVerifierStoreService`** でこれを実装することが可能です。また、 **`SRPVerifierStore`** の Java プロパティーファイル実装を使用して MBean を実装することもできます。

SRPVerifierStore 実装

すべてのパスワードハッシュ情報がシリアライズされたオブジェクトのファイルとして使用できなければならないため、**`SRPVerifierStore`** インターフェースのデフォルト実装は実稼動システムでは推奨されません。

SRPVerifierStore 実装は、特定のユーザー名に対して **SRPVerifierStore.VerifierInfo** オブジェクトへのアクセスを提供します。SRP アルゴリズムが必要とするパラメーターを取得するため、**getUserVerifier(String)** メソッドはユーザー SRP セッションの最初に SRPService によって呼び出されます。

VerifierInfo オブジェクトの要素

username

認証に使用されるユーザー名またはユーザー ID です。

verifier

アイデンティティーの証拠としてユーザーが入力するパスワードの一方向ハッシュです。**org.jboss.security.Util** クラスにはパスワードハッシュ化アルゴリズムを実行する **calculateVerifier** メソッドが含まれています。出力パスワードは**H(salt | H(username | ':' | password))** の形式を取ります。 **H** は RFC2945 で定義されている SHA セキュアハッシュ関数になります。ユーザー名は UTF-8 エンコーディングを使用して文字列から byte[] へ変換されます。

salt

データベースの情報が漏えいされた場合に、ベリファイアーパスワードデータベース上での総当たり辞書攻撃を難しくするために使用される乱数です。ユーザーの既存のクリアテキストパスワードがハッシュ化される時に、暗号強度が高い乱数アルゴリズムより値が生成されなければなりません。

g

SRP アルゴリズムプリミティブジェネレーターです。ユーザーごとの設定ではなく、既知の固定パラメーターとなります。 **org.jboss.security.srp.SRPConf** ユーティリティクラスは **g** の設定を複数提供します。 これには **SRPConf.getDefaultParams().g()** により取得される適切なデフォルトなどが含まれます。

N

SRP アルゴリズムセーフプライムモジュールです。ユーザーごとの設定ではなく、既知の固定パラメーターとなります。 org.jboss.security.srp.SRPConf ユーティリティクラスは **org.jboss.security.srp.SRPConf** は N の設定を複数提供します。これには **SRPConf.getDefaultParams().N()** より取得される適切なデフォルトなどが含まれます。

例14.14 SRPVerifierStore インターフェース

```
package org.jboss.security.srp;

import java.io.IOException;
import java.io.Serializable;
import java.security.KeyException;

public interface SRPVerifierStore
{
    public static class VerifierInfo implements Serializable
    {

        public String username;
```

```
        public byte[] salt;
        public byte[] g;
        public byte[] N;
    }


    public VerifierInfo getUserVerifier(String username)
        throws KeyException, IOException;

    public void setUserVerifier(String username, VerifierInfo info)
        throws IOException;


     public void verifyUserChallenge(String username, Object
auxChallenge)
         throws SecurityException;
}
```

バグを報告する

14.7. 機密性の高い文字列のパスワードボールト

14.7.1. クリアテキストファイルでの機密性が高い文字列のセキュア化について

Web アプリケーションと他のデプロイメントには、パスワードや他の機密性が高い文字列のような機密性が高い情報を含む XML デプロイメント記述子などのクリアテキストファイルが含まれることがよくあります。JBoss Enterprise Application Platform には、機密性が高い文字列を暗号化し、暗号化キーストアに格納できるパスワード vault メカニズムが含まれます。vault メカニズムは、セキュリティードメイン、セキュリティー領域、または他の検証システムで使用する文字列の復号化を管理します。これにより、セキュリティーのレイヤーが追加されます。このメカニズムは、サポートされるすべての Java Development Kit (JDK) 実装に含まれるツールに依存します。

バグを報告する

14.7.2. 機密性が高い文字列を格納する Java キーストアの作成

要件

- **keytool** コマンドを使用出来る必要があります。これは Java Runtime Environment (JRE) により提供されます。このファイルのパスを見つけます。Red Hat Enterprise Linux では、これは **/usr/bin/keytool** にインストールされます。

手順14.6 Java キーストアの設定

1. **キーストアと他の暗号化された情報を格納するディレクトリーを作成します。**
   キーストアと他の重要な情報を保持するディレクトリーを作成します。この残りの手順では、ディレクトリーが **/home/*USER*/vault/** であることを前提とします。

2. **keytool で使用するパラメーターを決定します。**
   以下のパラメーターを決定します。

alias

エイリアスは資格情報コンテナまたはキーストアに格納された vault または他のデータの一意の ID です。この手順の最後にあるコマンド例のエイリアスは **vault** です。エイリアスは大文字と小文字を区別します。

keyalg

暗号化に使用するアルゴリズム。デフォルト値は **DSA** です。この手順の例では **RSA** です。利用可能な他の選択肢については、JRE およびオペレーティングシステムのドキュメンテーションを参照してください。

keysize

暗号化キーのサイズにより、ブルートフォース攻撃により復号化する困難さが影響を受けます。キーのデフォルトサイズは 1024 です。これは 512 〜 1024 の範囲にあり、64 の倍数である必要があります。この手順の例では **1024** を使用します。

keystore

暗号化された情報と暗号化方法に関する情報を保持するデータベースのキーストア。キーストアを指定しない場合、使用するデフォルトのキーストアはホームディレクトリーの **.keystore** という名前のファイルです。これは、キーストアにデータを初めて追加したときに作成されます。この手順の例では、**vault.keystore** キーストアを使用します。

keystore コマンドには他の多くのオプションがあります。詳細については、JRE またはオペレーティングシステムのドキュメンテーションを参照してください。

3. **keystore コマンドが尋ねる質問の回答を決定します。**
keystore は、キーストアエントリーに値を入力するために次の情報を必要とします。

キーストアパスワード

キーストアを作成する場合は、パスワードを設定する必要があります。将来キーストアを使用するために、パスワードを提供する必要があります。覚えやすい強度の高いパスワードを作成します。キーストアは、パスワードや、キーストアが存在するファイルシステムおよびオペレーティングシステムのセキュリティーと同程度にセキュアです。

キーパスワード (任意設定)

キーストアパスワードに加え、保持する各キーにパスワードを指定することが可能です。このようなキーを使用するには、使用するたびにパスワードを提供する必要があります。通常、このファシリティーは使用されません。

名前 (名) と 名字 (姓)

この情報と一覧の他の情報は、一意にキーを識別して他のキーの階層に置くのに役立ちます。名前である必要はありませんが、キーに一意な 2 つの言葉である必要があります。この手順の例では、**Accounting Administrator** を使用します。これが証明書の*コモンネーム*になります。

組織単位

証明書を使用する人物を特定する単一の言葉です。アプリケーションユニットやビジネスユニットである場合もあります。この手順の例では **AccountingServices** を使用します。通常、1 つのグループやアプリケーションによって使用されるキーストアはすべて同じ組織単位を使用します。

組織

通常、所属する組織名を表す単一の言葉になります。一般的に、1 つの組織で使用されるすべての証明書で同じになります。この例では **MyOrganization** を使用します。

市または自治体

お住まいの市名。

州または県

お住まいの州や県または同等の行政区画。

国

2 文字の国コード。

これらすべての情報によってキーストアや証明書の階層が作成され、一貫性のある一意な名前付け構造が確実に使用されるようにします。

4. **keytool** コマンドを実行し、収集した情報を提供します。

例14.15 keystore コマンドの入出力例

```
$ keytool -genkey -alias vault -keyalg RSA -keysize 1024 -keystore
/home/USER/vault/vault.keystore
Enter keystore password: vault22
Re-enter new password:vault22
What is your first and last name?
  [Unknown]:  Accounting Administrator
What is the name of your organizational unit?
  [Unknown]:  AccountingServices
What is the name of your organization?
  [Unknown]:  MyOrganization
What is the name of your City or Locality?
  [Unknown]:  Raleigh
What is the name of your State or Province?
  [Unknown]:  NC
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=Accounting Administrator, OU=AccountingServices,
O=MyOrganization, L=Raleigh, ST=NC, C=US correct?
  [no]:  yes

Enter key password for <vault>
        (RETURN if same as keystore password):
```

結果

/home/*USER*/vault/ ディレクトリに **vault.keystore** という名前のファイルが作成されます。JBoss Enterprise Application Platform のパスワードなど、暗号化された文字列を格納するため使用される **vault** という 1 つのキーがこのファイルに保存されます。

バグを報告する

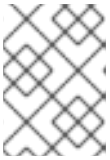
14.7.3. キーストアパスワードのマスキングとパスワード vault の初期化

要件

- 「機密性が高い文字列を格納する Java キーストアの作成」
- ***EAP_HOME*/bin/vault.sh** アプリケーションはコマンドラインインターフェースからアクセスできる必要があります。

1. **vault.sh コマンドを実行します。**
   ***EAP_HOME*/bin/vault.sh** を実行します。**0** を入力して新しい対話セッションを開始します。

2. **暗号化されたファイルが保存されるディレクトリを入力します。**
   このディレクトリはある程度保護されている必要がありますが、JBoss Enterprise Application Platform がアクセスできなければなりません。「機密性が高い文字列を格納する Java キーストアの作成」 の手順に従うと、キーストアはホームディレクトリにある**vault/** というディレクトリの中にあります。この例では **/home/*USER*/vault/** を使用します。

   注記

   必ずディレクトリ名の最後にスラッシュが含まれるようにしてください。ご使用のオペレーティングシステムに応じて / または \ を使用します。

3. **キーストアへのパスを入力します。**
   キーストアファイルへの完全パスを入力します。この例では **/home/*USER*/vault/vault.keystore** を使用します。

4. **キーストアパスワードを暗号化します。**
   次の手順に従って、設定ファイルやアプリケーションで安全に使用できるようキーストアのパスワードを暗号化します。

   a. **キーストアパスワードを入力します。**
      入力を促されたらキーストアのパスワードを入力します。

   b. **salt 値を入力します。**
      8 文字の salt 値を入力します。salt 値は反復回数(下記) と共にハッシュ値の作成に使用されます。

   c. **反復回数を入力します。**
      反復回数の値を入力します。

   d. **マスクされたパスワード情報を書き留めておきます。**
      マスクされたパスワードと salt、反復回数は標準出力へ書き出されます。これらの情報を安全な場所に書き留めておきます。攻撃者がこれらの情報を使用してパスワードを復号化する可能性があるからです。

   e. **vault のエイリアスを入力します。**
      入力を促されたら、vault のエイリアスを入力します。「機密性が高い文字列を格納する Java キーストアの作成」 に従って vault を作成した場合、エイリアスは **vault** になります。

5. **対話コンソールを終了します。**
   exit を入力して対話コンソールを終了します。

結果

設定ファイルとデプロイメントで使用するため、キーストアパスワードがマスキングされます。また、vault が完全設定され、すぐ使用できる状態になります。

バグを報告する

14.7.4. パスワード vault を使用するよう JBoss Enterprise Application Platform を設定

概要

設定ファイルにあるパスワードや機密性の高いその他の属性をマスキングする前に、これらを保存し復号化するパスワード vault を JBoss Enterprise Application Platform が認識するようにする必要があります。次の手順に従ってこの機能を有効にします。

要件

- 「機密性が高い文字列を格納する Java キーストアの作成」
- 「キーストアパスワードのマスキングとパスワード vault の初期化」

手順14.7 パスワード vault の設定

1. **コマンドの適切な値を決定します。**
   キーストアの作成に使用されるコマンドによって決定される以下のパラメーターの値を決定します。キーストア作成の詳細は 「機密性が高い文字列を格納する Java キーストアの作成」および 「キーストアパスワードのマスキングとパスワード vault の初期化」を参照してください。

パラメーター	説明
KEYSTORE_URL	ファイルシステムのパスまたはキーストアファイル。通常 `vault.keystore` のようになります。
KEYSTORE_PASSWORD	キーストアのアクセスに使用されるパスワード。この値はマスクされる必要があります。
KEYSTORE_ALIAS	キーストアの名前。
SALT	キーストアの値を暗号化および復号化するために使用されるソルト。
ITERATION_COUNT	暗号化アルゴリズムが実行される回数。
ENC_FILE_DIR	キーストアコマンドが実行されるディレクトリへのパス。通常、パスワード vault が含まれるディレクトリになります。
host (管理対象ドメインのみ)	設定するホストの名前。

2. **管理 CLI を使用してパスワード vault を有効にします。**

次のコマンドの1つを実行します。実行するコマンドは、管理対象ドメインを使用するか、スタンドアロンサーバー設定を使用するかによって異なります。コマンドの値は、手順の最初で使用した値に置き換えます。

- **管理対象ドメイン**

```
/host=YOUR_HOST/core-service=vault:add(vault-options=
[("KEYSTORE_URL" => "PATH_TO_KEYSTORE"), ("KEYSTORE_PASSWORD" =>
"MASKED_PASSWORD"), ("KEYSTORE_ALIAS" => "ALIAS"), ("SALT" =>
"SALT"),("ITERATION_COUNT" => "ITERATION_COUNT"), ("ENC_FILE_DIR"
=> "ENC_FILE_DIR")])
```

- **スタンドアロンサーバー**

```
/core-service=vault:add(vault-options=[("KEYSTORE_URL" =>
"PATH_TO_KEYSTORE"), ("KEYSTORE_PASSWORD" => "MASKED_PASSWORD"),
("KEYSTORE_ALIAS" => "ALIAS"), ("SALT" => "SALT"),
("ITERATION_COUNT" => "ITERATION_COUNT"), ("ENC_FILE_DIR" =>
"ENC_FILE_DIR")])
```

仮の値を用いたコマンドの例は次のとおりです。

```
/core-service=vault:add(vault-options=[("KEYSTORE_URL" =>
"/home/user/vault/vault.keystore"), ("KEYSTORE_PASSWORD" => "MASK-
3y28rCZlcKR"), ("KEYSTORE_ALIAS" => "vault"), ("SALT" =>
"12438567"),("ITERATION_COUNT" => "50"), ("ENC_FILE_DIR" =>
"/home/user/vault/")])
```

結果

パスワード vault を使用してマスキングされた文字列を復号化するよう JBoss Enterprise Application Platform が設定されます。vault に文字列を追加し、設定で使用する場合は 「Java キーストアに暗号化された機密性の高い文字列の保存および読み出し」 を参照してください。

バグを報告する

14.7.5. Java キーストアに暗号化された機密性の高い文字列の保存および読み出し

概要

パスワードや、機密性の高いその他の文字列が平文の設定ファイルに含まれるのはセキュアではありません。JBoss Enterprise Application Platform には、このような機密性の高い文字列をマスキングして暗号化されたキーストアに保存する機能や、設定ファイルでマスクされた値を使用する機能が含まれています。

要件

- 「機密性が高い文字列を格納する Java キーストアの作成」
- 「キーストアパスワードのマスキングとパスワード vault の初期化」
- 「パスワード vault を使用するよう JBoss Enterprise Application Platform を設定」
- ***EAP_HOME*/bin/vault.sh** アプリケーションはコマンドラインインターフェースからアクセスできる必要があります。

手順14.8 Java キーストアの設定

1. **vault.sh コマンドを実行します。**
   ***EAP_HOME*/bin/vault.sh** を実行します。**0** を入力して新しい対話セッションを開始します。

2. **暗号化されたファイルが保存されるディレクトリを入力します。**
   「機密性が高い文字列を格納する Java キーストアの作成」に記載された手順に従って作業を行った場合、キーストアはホームディレクトリの **vault/** というディレクトリにあります。ほとんどの場合では、暗号化されたすべての情報をキーストアとして同じ場所に保存するのが普通です。この例では **/home/*USER*/vault/** ディレクトリを使用します。

   注記

   必ずディレクトリ名の最後にスラッシュが含まれるようにしてください。ご使用のオペレーティングシステムに応じて **/** または **** を使用します。

3. **キーストアへのパスを入力します。**
   キーストアファイルへの完全パスを入力します。この例では **/home/*USER*/vault/vault.keystore** を使用します。

4. **キーストアパスワード、vault 名、ソルト、反復回数を入力します。**
   入力を促されたら、キーストアパスワード、vault 名、ソルト、反復回数を入力します。ハンドシェイクが実行されます。

5. **パスワードを保存するオプションを選択します。**
   オプション **0** を選択して、パスワードや機密性の高い他の文字列を保存します。

6. **値を入力します。**
   入力を促されたら、値を 2 回入力します。値が一致しない場合は再度入力するよう要求されます。

7. **vault ブロックを入力します。**
   同じリソースに関連する属性のコンテナである vault ブロックを入力します。属性名の例としては **ds_ExampleDS** などが挙げられます。データソースまたは他のサービス定義で、暗号化された文字列への参照の一部を形成します。

8. **属性名を入力します。**
   保存する属性の名前を入力します。 **password** が属性名の例の 1 つになります。

   結果

   属性が保存されたことが、以下のようなメッセージによって示されます。

   ```
   Attribute Value for (ds_ExampleDS, password) saved
   ```

9. **暗号化された文字列に関する情報を書き留めます。**
   メッセージは vault ブロック、属性名、共有キー、および設定で文字列を使用する場合のアドバイスを表示する標準出力を出力します。安全な場所にこの情報を書き留めておくようにしてください。出力例は次のとおりです。

   ```
   ************************************************
   Vault Block:ds_ExampleDS
   Attribute Name:password
   ```

```
Shared
Key:N2NhZDYzOTMtNWE0OS00ZGQ0LWE4MmEtMWNlMDMyNDdmNmI2TElORV9CUkVBS3Zh
dWx0
Configuration should be done as follows:
VAULT::ds_ExampleDS::password::N2NhZDYzOTMtNWE0OS00ZGQ0LWE4MmEtMWNlM
DMyNDdmNmI2TElORV9CUkVBS3ZhdWx0
************************************************
```

10. **設定で暗号化された文字列を使用します。**
プレーンテキストの文字列の代わりに、前の設定手順の文字列を使用します。以下は、上記の暗号化されたパスワードを使用するデータソースになります。

```
...
  <subsystem xmlns="urn:jboss:domain:datasources:1.0">
    <datasources>
      <datasource jndi-name="java:jboss/datasources/ExampleDS"
enabled="true" use-java-context="true" pool-name="H2DS">
        <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-
1</connection-url>
        <driver>h2</driver>
        <pool></pool>
        <security>
          <user-name>sa</user-name>

<password>${VAULT::ds_ExampleDS::password::N2NhZDYzOTMtNWE0OS00ZGQ0L
WE4MmEtMWNlMDMyNDdmNmI2TElORV9CUkVBS3ZhdWx0}</password>
        </security>
      </datasource>
      <drivers>
         <driver name="h2" module="com.h2database.h2">
            <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-
datasource-class>
         </driver>
      </drivers>
    </datasources>
  </subsystem>
...
```

式が許可されるドメインまたはスタンドアロン設定ファイルであれば、どこでも暗号化された文字列を使用することができます。

注記

特定のサブシステム内で式が許可されるかを確認するには、そのサブシステムに対して次の CLI コマンドを実行します。

```
/host=master/core-service=management/security-
realm=TestRealm:read-resource-description(recursive=true)
```

このコマンドの出力で、**expressions-allowed** パラメーターの値を探します。値が **true** であればこのサブシステムの設定内で式を使用できます。

文字列をキーストアに格納した後、次の構文を使用してクリアテキストの文字列を暗号化された文字列に置き換えます。

```
${VAULT::<replaceable>VAULT_BLOCK</replaceable>::
<replaceable>ATTRIBUTE_NAME</replaceable>::
<replaceable>ENCRYPTED_VALUE</replaceable>}
```

実環境の値の例は次のとおりです。vault ブロックは **ds_ExampleDS**、属性は **password** です。

```
<password>${VAULT::ds_ExampleDS::password::N2NhZDYzOTMtNWE0OS00ZGQ0LWE4MmEtMWNlMDMyNDdmNmI2TElORV9CUkVBS3ZhdWx0}</password>
```

バグを報告する

14.7.6. アプリケーションで機密性の高い文字列を保存および解決

概要

JBoss Enterprise Application Platform の設定要素は、セキュリティー vault メカニズムを通じて Java キーストアに保存される値に対して暗号化された文字列を解決する機能をサポートしています。この機能に対するサポートを独自のアプリケーションに追加することができます。

最初に、vault にパスワードを追加します。次に、クリアテキストのパスワードを vault に保存されているパスワードに置き換えます。この方法を使用してアプリケーションの機密性の高い文字列を分かりにくくすることができます。

要件

この手順を実行する前に、vault ファイルを格納するディレクトリが存在することを確認してください。JBoss Enterprise Application Platform を実行するユーザーが vault ファイルを読み書きできるパーミッションを持っていれば、vault ファイルの場所はどこでも構いません。この例では、**vault/** ディレクトリを **/home/*USER*/vault/** ディレクトリに置きます。vault 自体は **vault/** ディレクトリの中にある **vault.keystore** と呼ばれるファイルになります。

例14.16 vault へのパスワードの文字列の追加

***EAP_HOME*/bin/vault.sh** コマンドを用いて文字列を vault へ追加します。次の画面出力にコマンドと応答がすべて含まれています。ユーザー入力の値は強調文字で表されています。出力の一部は書式上、削除されています。Microsoft Windows ではコマンド名は **vault.bat** になります。Microsoft Windows のファイルパスでは、ディレクトリの分離記号として / ではなく \ が使用されることに注意してください。

```
[user@host bin]$ ./vault.sh
*********************************
****  JBoss Vault ********
*********************************
Please enter a Digit::   0: Start Interactive Session  1: Remove
Interactive Session  2: Exit
0
Starting an interactive session
Enter directory to store encrypted files:/home/user/vault/
Enter Keystore URL:/home/user/vault/vault.keystore
Enter Keystore password: ...
Enter Keystore password again: ...
```

```
Values match
Enter 8 character salt:12345678
Enter iteration count as a number (Eg: 44):25

Enter Keystore Alias:vault
Vault is initialized and ready for use
Handshake with Vault complete
Please enter a Digit::   0: Store a password  1: Check whether password
exists  2: Exit
0
Task:  Store a password
Please enter attribute value: sa
Please enter attribute value again: sa
Values match
Enter Vault Block:DS
Enter Attribute Name:thePass
Attribute Value for (DS, thePass) saved

Please make note of the following:
********************************************
Vault Block:DS
Attribute Name:thePass
Shared
Key:OWY5M2I5NzctYzdkOS00MmZhLWExZGYtNjczM2U5ZGUyOWIxTElORV9CUkVBS3ZhdWx0
Configuration should be done as follows:
VAULT::DS::thePass::OWY5M2I5NzctYzdkOS00MmZhLWExZGYtNjczM2U5ZGUyOWIxTElO
RV9CUkVBS3ZhdWx0
********************************************

Please enter a Digit::   0: Store a password  1: Check whether password
exists  2: Exit
2
```

Java コードに追加される文字列は、出力の最後の値である **VAULT** で始まる行です。

次のサーブレットは、クリアテキストのパスワードの代わりに vault された文字列を使用します。違いを確認できるようにするため、クリアテキストのパスワードはコメントアウトされています。

例14.17 vault されたパスワードを使用するサーブレット

```
package vaulterror.web;

import java.io.IOException;
import java.io.Writer;

import javax.annotation.Resource;
import javax.annotation.sql.DataSourceDefinition;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
```

```
/*@DataSourceDefinition(
        name = "java:jboss/datasources/LoginDS",
        user = "sa",
        password = "sa",
        className = "org.h2.jdbcx.JdbcDataSource",
        url = "jdbc:h2:tcp://localhost/mem:test"
)*/
@DataSourceDefinition(
        name = "java:jboss/datasources/LoginDS",
        user = "sa",
        password =
"VAULT::DS::thePass::OWY5M2I5NzctYzdkOS00MmZhLWExZGYtNjczM2U5ZGUyOWIxTEl
ORV9CUkVBS3ZhdWx0",
        className = "org.h2.jdbcx.JdbcDataSource",
        url = "jdbc:h2:tcp://localhost/mem:test"
)
@WebServlet(name = "MyTestServlet", urlPatterns = { "/my/" },
loadOnStartup = 1)
public class MyTestServlet  extends HttpServlet {

    private static final long serialVersionUID = 1L;


    @Resource(lookup = "java:jboss/datasources/LoginDS")
    private DataSource ds;

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
        Writer writer = resp.getWriter();
        writer.write((ds != null) + "");
    }
}
```

これでサーブレットが vault された文字列を解決できるようになります。

バグを報告する

14.8. JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS)

14.8.1. JACC (Java Authorization Contract for Containers) について

JACC (Java Authorization Contract for Containers) はコンテナと承認サービスプロバイダー間のインターフェースを定義する規格で、これによりコンテナによって使用されるプロバイダーの実装が可能になります。JACC は JSR-115 に定義されており、http://jcp.org/en/jsr/detail?id=115 の Java Community Process Web サイトで確認できます。Java EE バージョン 1.3 より、コアの Java Enterprise Edition (Java EE) 仕様の一部となっています。

JBoss Enterprise Application Platform は、セキュリティーサブシステムのセキュリティー機能内に JACC のサポートを実装します。

バグを報告する

14.8.2. JACC (Java Authorization Contract for Containers) のセキュリティーの設定

JACC (Java Authorization Contract for Containers) を設定するには、適切なモジュールでセキュリティードメインを設定し、適切なパラメーターが含まれるよう **jboss-web.xml** を編集する必要があります。

セキュリティードメインへの JACC サポートの追加

セキュリティードメインに JACC サポートを追加するには、**required** フラグセットで **JACC** 承認ポリシーをセキュリティードメインの承認スタックへ追加します。以下は JACC サポートを持つセキュリティードメインの例になりますが、セキュリティードメインは直接 XML には設定されず、管理コンソールまたは管理 CLI で設定されます。

```
<security-domain name="jacc" cache-type="default">
    <authentication>
        <login-module code="UsersRoles" flag="required">
        </login-module>
    </authentication>
    <authorization>
        <policy-module code="JACC" flag="required"/>
    </authorization>
</security-domain>
```

JACC を使用するよう Web アプリケーションを設定

jboss-web.xml はデプロイメントの **META-INF/** または **WEB-INF/** ディレクトリに存在し、Web コンテナに対する追加の JBoss 固有の設定を格納し、上書きします。JACC が有効になっているセキュリティードメインを使用するには、**<security-domain>** 要素が含まれるようにし、さらに **<use-jboss-authorization>** 要素を **true** に設定する必要があります。以下は、上記の JACC セキュリティードメインを使用するよう適切に設定されているアプリケーションになります。

```
<jboss-web>
    <security-domain>jacc</security-domain>
    <use-jboss-authorization>true</use-jboss-authorization>
</jboss-web>
```

JACC を使用するよう EJB アプリケーションを設定

セキュリティードメインと JACC を使用するよう EJB を設定する方法は Web アプリケーションの場合とは異なります。EJB の場合、**ejb-jar.xml** 記述子にてメソッドまたはメソッドのグループ上で*メソッドパーミッション*を宣言できます。**<ejb-jar>** 要素内では、すべての子 **<method-permission>** 要素に JACC ロールに関する情報が含まれます。詳細は設定例を参照してください。**EJBMethodPermission** クラスは Java Enterprise Edition 6 API の一部で、http://docs.oracle.com/javaee/6/api/javax/security/jacc/EJBMethodPermission.html で説明されています。

例14.18 EJB の JACC メソッドパーミッション例

```
<ejb-jar>
  <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>
</ejb-jar>
```

Web アプリケーションと同様にセキュリティードメインを使用して EJB の認証および承認メカニズムを指定することも可能です。セキュリティードメインは **<security>** 子要素の **jboss-ejb3.xml** 記述子に宣言されます。セキュリティードメインの他に、EJB が実行されるプリンシパルを変更する *run-as プリンシパル*を指定することもできます。

例14.19 EJB におけるセキュリティードメイン宣言の例

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

バグを報告する

14.9. JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS)

14.9.1. JASPI (Java Authentication SPI for Containers) のセキュリティーについて

Java Application SPI for Containers (JASPI または JASPIC) は Java アプリケーションのプラグ可能なインターフェースです。Java Community Process の JSR-196 に定義されています。この仕様の詳細は http://www.jcp.org/en/jsr/detail?id=196 を参照してください。

バグを報告する

14.9.2. JASPI (Java Authentication SPI for Containers) のセキュリティーの設定

JASPI プロバイダーに対して認証するには、**<authentication-jaspi>** 要素をセキュリティードメインに追加します。設定は標準的な認証モジュールと似ていますが、ログインモジュール要素は **<login-module-stack>** 要素で囲まれています。設定の構成は次のとおりです。

例14.20 authentication-jaspi 要素の構成

```
<authentication-jaspi>
 <login-module-stack name="...">
   <login-module code="..." flag="...">
     <module-option name="..." value="..."/>
   </login-module>
 </login-module-stack>
```

```
 <auth-module code="..." login-module-stack-ref="...">
   <module-option name="..." value="..."/>
 </auth-module>
</authentication-jaspi>
```

ログインモジュール自体は標準的な認証モジュールと全く同じように設定されます。

Web ベースの管理コンソールは JASPI 認証モジュールの設定を公開しないため、JBoss Enterprise Application Platform を完全に停止してから、設定を ***EAP_HOME*/domain/configuration/domain.xml** または ***EAP_HOME*/standalone/configuration/standalone.xml** へ直接追加する必要があります。

バグを報告する

第15章 シングルサインオン (SSO)

15.1. WEB アプリケーションのシングルサインオン (SSO) について

概要

*SSO (シングルサインオン)*は 1 つのリソースへの認証を用いて他のリソースへのアクセスを暗黙的に承認できるようにします。

クラスター化された SSO とクラスター化されていない SSO

クラスター化されていない SSO は、アプリケーションの承認情報の共有を同じ仮想ホスト内に制限します。また、ホストの障害に対する耐性を持ちません。クラスター化された SSO データは複数の仮想ホストのアプリケーション間で共有することができ、フェイルオーバーに対する耐性を持ちます。さらに、クラスター化された SSO はロードバランサーからのリクエストを受信することができます。

SSO の仕組み

リソースが保護されていない場合、ユーザーの認証は完全に無視されます。ユーザーが保護されたリソースにアクセスすると、ユーザーの認証が必要になります。

認証に成功すると、ユーザーに関連するロールが保存され、関連する他のリソースすべての承認に使用されます。

ユーザーがアプリケーションからログアウトしたり、アプリケーションがプログラムを用いてセッションを無効化した場合、永続化された承認データはすべて削除され、プロセスを最初からやり直しします。

他のセッションが有効である場合、セッションタイムアウトは SSO セッションを無効化しません。

SSO の制限

サードパーティー境界にまたがる伝搬がない

JBoss Enterprise Application Platform のコンテナ内にデプロイされたアプリケーションの間でのみ SSO を使用できます。

コンテナ管理の認証のみ使用可能

アプリケーションの **web.xml** で **<login-config>** などのコンテナ管理認証要素を使用しなければなりません。

クッキーが必要

SSO はブラウザークッキーを介して維持されます。URL の再書き込みはサポートされていません。

レルムとセキュリティードメインの制限

requireReauthentication パラメーターが **true** に設定されている場合を除き、同じ SSO バルブに設定されたすべての Web アプリケーションは、**web.xml** の同じレルム設定と同じセキュリティードメインを共有しなければなりません。

関与する Web アプリケーションの 1 つに対し、Host 要素内または Engine 要素周囲で Realm 要素をネストできますが、context.xml 要素内で Realm 要素はネストできません。

jboss-web.xml に設定された **<security-domain>** はすべての Web アプリケーション全体で一貫していなければなりません。

すべてのセキュリティー統合が同じ認証情報 (ユーザー名やパスワードなど) を許可しなければなりません。

バグを報告する

15.2. WEB アプリケーションのクラスター化されたシングルサインオン (SSO) について

シングルサインオン (SSO) とは、ユーザーが単一の Web アプリケーションへ認証を行い、認証に成功した場合は複数の他のアプリケーションに承認が与えられる機能のことです。クラスター化された SSO はクラスター化されたキャッシュに認証および承認情報を保存します。これにより、複数の異なるサーバー上にあるアプリケーションが情報を共有し、ホストの 1 つが障害を起こした場合でも情報が障害に耐えられるようにします。

SSO の設定はバルブと呼ばれます。バルブは、サーバーやサーバーグループのレベルに設定されるセキュリティードメインへ接続されます。キャッシュされた同じ認証情報を共有する必要がある各アプリケーションは同じバルブを使用するよう設定されます。これは、アプリケーションの **`jboss-web.xml`** に設定されます。

JBoss Enterprise Application Platform の Web サブシステムによってサポートされる一般的な SSO バルブの一部は次の通りです。

- Apache Tomcat の ClusteredSingleSignOn
- Apache Tomcat の IDPWebBrowserSSOValve
- PicketLink によって提供される SPNEGO ベースの SSO

バルブのタイプによっては、バルブが適切に動作するよう、セキュリティードメインに追加設定を行う必要がある場合があります。

バグを報告する

15.3. 適切な SSO 実装の選択

JBoss Enterprise Application Platform は Web アプリケーションや EJB アプリケーション、Web サービスなどの Java Enterprise Edition (EE) アプリケーションを実行します。SSO (Single Sign On: シングルサインオン) により、これらのアプリケーションの間でセキュリティーコンテキストとアイデンティティー情報が伝播できるようになります。組織のニーズに合わせ、異なる SSO ソリューションを使用することができます。使用するソリューションは以下の状況により異なります。 1) Web アプリケーションや EJBアプリケーション、Web サービスのどれを使用するか。 2) アプリケーションが同じサーバー、複数のクラスター化されていないサーバー、複数のクラスター化されたサーバーのどれを使用するか。 3) デスクトップベースの認証システムに統合する必要があるかまたはアプリケーション間でのみ認証が必要になるか。

Kerberos ベースのデスクトップ SSO

Microsoft Active Directory など、Kerberos ベースの認証承認システムがすでに組織で使用されている場合は、同じシステムを使用して JBoss Enterprise Application Platform 上で実行されているエンタープライズアプリケーションを透過的に認証することができます。

クラスター化されていない Web アプリケーション SSO

同じサーバーグループやインスタンス内で実行するアプリケーション間でセキュリティー情報を伝播する必要がある場合、クラスター化されていない SSO を使用することができます。この場合、アプリケーションの **jboss-web.xml** 記述子にバルブを設定することのみが必要となります。

クラスター化された Web アプリケーション SSO

複数の JBoss Enterprise Application Platform インスタンス全体のクラスター化された環境で実行されるアプリケーションの間でセキュリティー情報を伝播する必要がある場合、クラスター化された SSO バルブを使用することができます。このバルブはアプリケーションの **jboss-web.xml** に設定されます。

バグを報告する

15.4. WEB アプリケーションでの SSO (シングルサインオン) の使用

概要

SSO (シングルサインオン) の機能は Web および Infinispan サブシステムによって提供されます。この手順に従って Web アプリケーションに SSO を設定します。

要件

- 認証と承認を処理するセキュリティードメインが設定されている必要があります。
- **infinispan** サブシステムが存在する必要があります。管理対象ドメインの場合、このサブシステムは **full-ha** プロファイルにあります。スタンドアロンサーバーでは**standalone-full-ha.xml** 設定を使用します。
- **webcache-container** と SSO cache-container が存在する必要があります。例の設定ファイルには **web** cache-container がすでに含まれており、一部の設定には SSO cache-container も含まれています。以下のコマンドを使用して SSO キャッシュコンテナをチェックし、有効にします。これらのコマンドは管理対象ドメインの **full** プロファイルを変更することに注意してください。スタンドアロンサーバーに対して異なるプロファイルを使用したり、コマンドの **/profile=full** 部分を削除するため、コマンドを変更することもできます。

> **例15.1 web cache-container の確認**
>
> 前述のプロファイルや設定には、デフォルトとして **web** cache-container が含まれています。次のコマンドを使用して、**web** cache-container の存在を確認します。異なるプロファイルを使用する場合は、**ha** をその名前に置き換えます。
>
> ```
> /profile=ha/subsystem=infinispan/cache-container=web/:read-
> resource(recursive=false,proxies=false,include-
> runtime=false,include-defaults=true)
> ```
>
> サブシステムが存在する場合、結果は **success** になります。存在しない場合は追加する必要があります。

> **例15.2 web cache-container の追加**
>
> 次の 3 つのコマンドを使用して **web** cache-container を設定に対して有効にします。必要に応じてプロファイルの名前やその他のパラメーターを変更します。以下のパラメーターはデフォルト設定で使用されるパラメーターになります。
>
> -

```
/profile=ha/subsystem=infinispan/cache-container=web:add(aliases=
["standard-session-cache"],default-
cache="repl",module="org.jboss.as.clustering.web.infinispan")
```

```
/profile=ha/subsystem=infinispan/cache-
container=web/transport=TRANSPORT:add(lock-timeout=60000)
```

```
/profile=ha/subsystem=infinispan/cache-container=web/replicated-
cache=repl:add(mode="ASYNC",batching=true)
```

例15.3 SSO cache-container の確認

次の管理 CLI コマンドを実行します。

```
/profile=ha/subsystem=infinispan/cache-container=web/:read-
resource(recursive=true,proxies=false,include-
runtime=false,include-defaults=true)
```

"sso" => { のような出力を探します。

このような出力が見つからない場合、設定に SSO cache-container は存在しません。

例15.4 SSO cache-container の追加

```
/profile=ha/subsystem=infinispan/cache-container=web/replicated-
cache=sso:add(mode="SYNC", batching=true)
```

- SSO を使用するよう **web** サブシステムを設定する必要があります。次のコマンドは、**default-host** という仮想サーバー上と、クッキードメイン **domain.com** で SSO を有効にします。キャッシュ名は **sso** で、再認証は無効になっています。

```
/profile=ha/subsystem=web/virtual-server=default-
host/sso=configuration:add(cache-container="web",cache-
name="sso",reauthenticate="false",domain="domain.com")
```

- SSO 情報を共有する各アプリケーションは、**jboss-web.xml** デプロイメント記述子にある同じ **<security-domain>** と **web.xml** 設定ファイルにある同じレルムを使用するよう設定されている必要があります。

クラスター化された SSO バルブとクラスター化されていない SSO バルブの違い

クラスター化された SSO では個別のホスト間で認証を共有できますが、クラスター化されていない SSO では共有できません。どちらの SSO も同じように設定されますが、クラスター化された SSO には永続データのクラスタリングレプリケーションを制御する **cacheConfig** や **processExpiresInterval**、 **maxEmptyLife** パラメーターが含まれています。

例15.5 クラスター化された SSO 設定の例

クラスター化された SSO とクラスター化されていない SSO は大変似ているため、クラスター化されている設定のみを取り上げます。この例は **tomcat** と呼ばれるセキュリティードメインを使用します。

```
<jboss-web>
 <security-domain>tomcat</security-domain>
 <valve>
  <class-
name>org.jboss.web.tomcat.service.sso.ClusteredSingleSignOn</class-name>
  <param>
   <param-name>maxEmptyLife</param-name>
   <param-value>900</param-value>
  </param>
 </valve>
</jboss-web>
```

表15.1 SSO 設定のオプション

オプション	説明
cookieDomain	SSO クッキーに使用するホストドメインです。デフォルトは **/** です。**app1.xyz.com** と **app2.xyz.com** によるクッキーの共有を許可するには、cookieDomain を **xyz.com** に設定します。
maxEmptyLife	クラスター化された SSO のみ設定可能です。失効する前に、アクティブなセッションを持たない SSO バルブを 1 つのリクエストが使用できる最大秒数。唯一バルブにアクティブなセッションが付加されている場合、正の値を設定するとノードのシャットダウンが適切に処理されるようになります。maxEmptyLife を **0** に設定すると、ローカルセッションがコピーされると同時にバルブが終了しますが、クラスター化されたアプリケーションからのセッションのバックアップコピーは他のクラスターノードが使用できるようになります。バルブの管理セッションの生存期間を越えてバルブが生存できるようにすると、他のリクエストを実行する時間がユーザーに与えられます。このリクエストはセッションのバックアップコピーをアクティベートする他のノードへフェイルオーバーすることができます。デフォルトは **1800** 秒 (30 分) です。
processExpiresInterval	クラスター化された SSO のみ設定可能です。**MaxEmptyLife** タイムアウトを失効した SSO インスタンスをバルブが発見し無効化する動作の間隔の最初秒数。デフォルトは **60** (1 分) です。
requiresReauthentication	true の場合、各リクエストはキャッシュされた認証情報を使用してセキュリティーレルムへ再認証します。false の場合 (デフォルト)、バルブによる新しい要求の認証には有効な SSO クッキーのみが必要になります。

セッションの無効化

アプリケーションはメソッド **`javax.servlet.http.HttpSession.invalidate()`** を呼び出し、プログラムを用いてセッションを無効化することができます。

バグを報告する

15.5. KERBEROS について

Kerberos はクライアント/サーバーアプリケーションのネットワーク認証プロトコルです。秘密鍵の対称暗号化を使用して、安全でないネットワーク全域で安全に認証を行えるようにします。

Kerberos はチケットと呼ばれるセキュリティートークンを使用します。安全なサービスを使用するには、ネットワークのサーバー上で稼働している TGS (チケット交付サービス: Ticket Granting Service) よりチケットを取得する必要があります。チケットの取得後、ネットワーク上で実行している別のサービスである AS (認証サービス: Authentication Service) より ST (サービスチケット: Service Ticket) を要求します。その後、ST を使用して使用したいサービスを認証します。TGS と AS は KDC (鍵配布センター: Key Distribution Center) と呼ばれるエンクロージングサービス内で実行されます。

Kerberos はクライアントサーバー環境で使用する目的で開発されているため、Web アプリケーションやシンクライアント環境ではほとんど使用されません。しかし、多くの組織で Kerberos システムはデスクトップの認証に使用されており、Web アプリケーション向けに別のシステムを作成せずに既存システムを再使用することが好まれます。Kerberos は Microsoft Active Directory には不可欠なもので、多くの Red Hat Enterprise Linux 環境でも使用されています。

バグを報告する

15.6. SPNEGO について

SPNEGO (Simple and Protected GSS_API Negotiation Mechanism) は Web アプリケーションで使用するため Kerberos ベースの SSO (Single Sign On) 環境を拡張するメカニズムを提供します。

Web ブラウザーなどのクライアントコンピューター上のアプリケーションが Web サーバーの保護ページにアクセスしようとすると、サーバーは承認が必要であることを伝えます。その後、アプリケーションは KDC (Kerberos Key Distribution Center) からのサービスチケットを要求します。チケットの取得後、アプリケーションはこのチケットをSPNEGO 向けにフォーマットされた要求にラップし、ブラウザーより Web アプリケーションへ返信します。デプロイされた Web アプリケーションを実行している Web コンテナが要求をアンパックし、チケットを認証します。認証に成功するとアクセスが許可されます。

SPNEGO は Red Hat Enterprise Linux に含まれる Kerberos サービスや Microsoft Active Directory には不可欠な Kerberos サーバーなど、全タイプの Kerberos プロバイダーと動作します。

バグを報告する

15.7. MICROSOFT ACTIVE DIRECTORY について

Microsoft Active Directory は Microsoft Windows のドメインでユーザーとコンピューターを認証するために Microsoft によって開発されたディレクトリサービスです。Microsoft Windows Server に含まれています。Microsoft Windows Server のコンピューターはドメインコントローラーと呼ばれます。Samba サービスを実行している Red Hat Enterprise Linux サーバーもこのようなネットワークでドメインコントローラーとして機能することが可能です。

Active Directory は連携する以下の 3 つのコア技術に依存します。

- ユーザーやコンピューター、パスワードなどのリソースの情報を保存する LDAP (Lightweight Directory Access Protocol) 。
- ネットワーク上で安全な認証を提供する Kerberos。
- IP アドレスやコンピューターのホスト名、ネットワーク上のその他のデバイス間でマッピングを提供する DNS (Domain Name Service)。

バグを報告する

15.8. WEB アプリケーションに対して KERBEROS または MICROSOFT ACTIVE DIRECTORY のデスクトップ SSO を設定する

はじめに

Microsoft Active Directory など、組織における既存の Kerberos ベースの認証承認インフラストラクチャーを使用して Web アプリケーションや EJB アプリケーションを認証するため、Enterprise Application Platform 6 に内蔵される JBoss Negotiation の機能を使用することが可能です。Web アプリケーションを適切に設定すれば、デスクトップまたはネットワークへのログインに成功するだけで Web アプリケーションに対して透過的な認証を行えるため、追加のログインプロンプトは必要ありません。

JBoss Enterprise Application Platform の以前のバージョンとの相違点

JBoss Enterprise Application Platform 6 と以前のバージョンには顕著な違いがいくつかあります。

- セキュリティードメインは、管理対象ドメインの各プロファイルまたは各スタンドアロンサーバーに対して設定されます。セキュリティードメインはデプロイメントの一部ではありません。デプロイメントが使用する必要のあるセキュリティードメインは、デプロイメントの **`jboss-web.xml`** または **`jboss-ejb3.xml`** ファイルに名前が指定されています。
- セキュリティープロパティーは設定の一部分で、セキュリティードメインの一部として設定されます。デプロイメントの一部ではありません。
- デプロイメントの一部としてオーセンティケーターを上書きすることができなくなりましたが、NegotiationAuthenticator バルブを **`jboss-web.xml`** 記述子に追加すると同じ結果を得ることができます。バルブでも **`<security-constraint>`** および **`<login-config>`** 要素が **`web.xml`** に定義されている必要があります。これらはセキュアなリソースを決定するために使用されますが、選択された auth-method は **`jboss-web.xml`** の NegotiationAuthenticator バルブによって上書きされます。
- セキュリティードメインの **`CODE`** 属性は、完全修飾クラス名ではなく、単純名を使用するようになりました。次の表は、これらのクラスと JBoss Negotiation に使用されるクラスとのマッピングを表しています。

表15.2 ログインモジュールコードとクラス名

単純名	クラス名	目的
Kerberos	com.sun.security.auth.module.Krb5LoginModule	Kerberos ログインモジュール。
SPNEGO	org.jboss.security.negotiation.spnego.SPNEGOLoginModule	Web アプリケーションが Kerberos 認証サーバーへ認証できるようにするメカニズム。

単純名	クラス名	目的
AdvancedLdap	org.jboss.security.negotiation.AdvancedLdapLoginModule	Microsoft Active Directory 以外の LDAP サーバーと使用されます。
AdvancedAdLdap	org.jboss.security.negotiation.AdvancedADLoginModule	Microsoft Active Directory の LDAP サーバーを使用されます。

Jboss Negotiation Toolkit

JBoss Negotiation Toolkit は https://community.jboss.org/servlet/JiveServlet/download/16876-2-34629/jboss-negotiation-toolkit.war よりダウンロード可能なデバッグ用のツールです。アプリケーションを実稼動環境に導入する前に認証メカニズムをデバッグし、テストできるようにするため提供されている追加のツールです。サポート対象のツールではありませんが、SPENEGO を Web アプリケーションに対して設定することは難しいこともあるため、大変便利なツールと言えます。

手順15.1 Web または EJB アプリケーションへ SSO 認証を設定

1. **サーバーのアイデンティティーを表すセキュリティードメインを 1 つ設定します。必要な場合はシステムプロパティーを設定します。**
   最初のセキュリティードメインは、コンテナ自体をディレクトリーサービスへ認証します。ユーザーではなくコンテナ自体の認証であるため、ある種の静的ログインメカニズムを受容するログインモジュールを使用する必要があります。この例では静的プリンシパルを使用し、認証情報が含まれるキータブファイルを参照します。

   明確にするため、この例では XML コードが提供されていますが、管理コンソールまたは管理 CLI を使用してセキュリティードメインを設定するようにしてください。

```
<security-domain name="host" cache-type="default">
   <authentication>
      <login-module code="Kerberos" flag="required">
         <module-option name="storeKey" value="true"/>
         <module-option name="useKeyTab" value="true"/>
         <module-option name="principal"
value="host/testserver@MY_REALM"/>
         <module-option name="keyTab"
value="/home/username/service.keytab"/>
         <module-option name="doNotPrompt" value="true"/>
         <module-option name="debug" value="false"/>
      </login-module>
   </authentication>
</security-domain>
```

2. **Web アプリケーションやアプリケーションをセキュアにするため、2 つ目のセキュリティードメインを設定します。必要な場合はシステムプロパティーを設定します。**
   2 つ目のセキュリティードメインは、個別のユーザーを Kerberos または SPNEGO 認証サーバーへ認証するために使用されます。ユーザーの認証に最低でも 1 つのログインモジュールが必要で、ユーザーに適用するロールを検索するために別のログインモジュールが必要となりま

す。次の XML コードは SPNEGO セキュリティードメインの例を表しています。これには、ロールを個別のユーザーにマッピングする承認モジュールが含まれます。認証サーバー上でロールを検索するモジュールを使用することもできます。

```
<security-domain name="SPNEGO" cache-type="default">
   <authentication>
      <!-- Check the username and password -->
      <login-module code="SPNEGO"  flag="requisite">
         <module-option name="password-stacking"
value="useFirstPass"/>
         <module-option name="serverSecurityDomain" value="host"/>
      </login-module>
      <!-- Search for roles -->
      <login-module code="UserRoles" flag="required">
         <module-option name="password-stacking"
value="useFirstPass" />
         <module-option name="usersProperties" value="spnego-
users.properties" />
         <module-option name="rolesProperties" value="spnego-
roles.properties" />
      </login-module>
   </authentication>
</security-domain>
```

3. **web.xml の security-constraint と login-config を指定します。**
   web.xml 記述子にはセキュリティー制約とログイン設定に関する情報が含まれています。セキュリティー制約とログイン情報の値の例は次の通りです。

```
<security-constraint>
   <display-name>Security Constraint on Conversation</display-name>
   <web-resource-collection>
      <web-resource-name>examplesWebApp</web-resource-name>
      <url-pattern>/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
   <role-name>RequiredRole</role-name>
   </auth-constraint>
</security-constraint>

<login-config>
   <auth-method>SPNEGO</auth-method>
   <realm-name>SPNEGO</realm-name>
</login-config>

<security-role>
   <description> role required to log in to the
Application</description>
   <role-name>RequiredRole</role-name>
</security-role>
```

4. **jboss-web.xml 記述子にセキュリティードメインと他の設定を指定します。**

クライアント側のセキュリティードメイン (例の 2 番目のセキュリティードメイン) の名前をデプロイメントの **jboss-web.xml** 記述子に指定し、アプリケーションがこのセキュリティードメインを使用するよう指示します。

オーセンティケーターを直接上書きすることができなくなりましたが、必要な場合は NegotiationAuthenticator をバルブとして **jboss-web.xml** 記述子に追加することができます。**<jacc-star-role-allow>** は任意で、複数のロール名を一致させるためアスタリスク (*) の使用を許可します。

```
<jboss-web>
   <security-domain>java:/jaas/SPNEGO</security-domain>
   <valve>
      <class-
name>org.jboss.security.negotiation.NegotiationAuthenticator</class-
name>
   </valve>
   <jacc-star-role-allow>true</jacc-star-role-allow>
</jboss-web>
```

5. **アプリケーションの MANIFEST.MF に依存関係を追加し、Negotiation クラスを見つけます。**
   Web アプリケーションが JBoss Negotiation クラスを見つけるには、クラス **org.jboss.security.negotiation** 上の依存関係をデプロイメントの **META-INF/MANIFEST.MF** マニフェストに追加する必要があります。適切にフォーマットされたエントリは次の通りです。

```
Manifest-Version: 1.0
Build-Jdk: 1.6.0_24
Dependencies: org.jboss.security.negotiation
```

結果

Web アプリケーションが Kerberos や Microsoft Active Directory、 SPNEGO 対応のディレクトリサービスに対して認証情報を許可し、認証します。既にディレクトリサービスにログインしているシステムよりユーザーがアプリケーションを実行し、必要なロールが既にユーザーに適用されている場合、Web アプリケーションは認証を要求しないため、SSO の機能が実現されます。

バグを報告する

第16章 コンテナインタープリター

16.1. コンテナーインターセプターについて

JSR 318, Enterprise JavaBeans 3.1 仕様で定義された標準的な Java EE 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伝播と、トランザクション管理、他のコンテナーにより提供された呼び出し処理を完了した後に実行されることが期待されます。これは、特定のコンテナー固有インターセプターが実行される前にユーザーアプリケーションが呼び出しをインターセプトする必要がある場合に問題となります。

JBoss Enterprise Application Platform 6.0 より前のリリースでは、サーバーサイドインターセプターを呼び出しフローに組み込み、コンテナーが呼び出し処理を完了する前にユーザーアプリケーション固有ロジックを実行することができました。JBoss Enterprise Application Platform 6.1 には、この機能が実装されるようになりました。この実装により、標準的な Java EE インターセプターをコンテナーインターセプターとして使用できるようになります (3.1 バージョンの ejb-jar デプロイメント記述子に対して `ejb-jar.xml` ファイルで許可されたのと同じ XSD 要素が使用されます)。

コンテナーインターセプターをインターセプターチェーンに配置

EJB に設定されたコンテナーインターセプターは、JBoss Enterprise Application Platform 6.1 がセキュリティーインターセプター、トランザクション管理インターセプター、他のサーバーにより提供されたインターセプターを提供する前に実行されることが保証されます。これにより、ユーザーアプリケーション固有コンテナーインターセプターは呼び出しが続行される前に関連するすべてのコンテキストデータを処理または設定できます。

コンテナーインターセプターと Java 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 Java EE インターセプターに似ていますが、API セマンティクスでいくつかの違いがあります。たとえば、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禁止されています。これは、EJB コンポーネントがセットアップまたはインスタンス化されるよりかなり前にこれらのインターセプターが呼び出されるためです。

バグを報告する

16.2. コンテナーインターセプタークラスの作成

概要

コンテナーインターセプタークラスは、単純な Plain Old Java Object (POJO) です。`@javax.annotation.AroundInvoke` 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指定します。

呼び出し用に `iAmAround` メソッドをマークするコンテナーインターセプタークラスの例は次のとおりです。

例16.1 コンテナーインターセプタークラスの例

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

```
        }
    }
```

このクラスを使用するよう設定されたコンテナーインターセプター記述子ファイルの例については、サンプル **jboss-ejb3.xml** ファイルを参照してください (「コンテナーインターセプターの設定」)。

バグを報告する

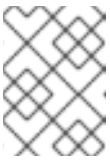
16.3. コンテナーインターセプターの設定

概要

コンテナーインターセプターは標準的な Java EE インターセプターライブラリーを使用します (つまり、3.1 バージョンの ejb-jar デプロイメント記述子用 **ejb-jar.xml** ファイルで許可されたのと同じ XSD 要素を使用します)。コンテナーインターセプターは標準的な Jave EE インターセプターライブラリーに基づくため、デプロイメント記述子を使用してのみ設定できます。これにより、アプリケーションは JBoss 固有のアノテーションまたは他のライブラリー依存関係を必要としなくなります。コンテナーインターセプターの詳細については、「コンテナーインターセプターについて」を参照してください。

手順16.1 記述子ファイルを作成してコンテナーインターセプターを設定

1. EJB デプロイメントの **META-INF** ディレクトリーで **jboss-ejb3.xml** ファイルを作成します。
2. 記述子ファイルで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、デプロイメントのすべての EJB にインターセプターをバインドします。
        - 特定の EJB 名を使用して個別 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、Java EE インターセプターの場合と同様に EJB 3.1 XSD を使用して設定されます。

3. 上記の要素の例として以下の記述ファイルを参照してください。

例16.2 jboss-ejb3.xml

```
<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.Cont
ainerInterceptorOne</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.Clas
sLevelContainerInterceptor</interceptor-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.Meth
odSpecificContainerInterceptor</interceptor-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.Clas
sLevelContainerInterceptor</interceptor-class>
                    <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.Meth
odSpecificContainerInterceptor</interceptor-class>
                    <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.Cont
ainerInterceptorOne</interceptor-class>
                </interceptor-order>
                <method>
                    <method-
name>echoInSpecificOrderOfContainerInterceptors</method-name>
                </method>
            </jee:interceptor-binding>
        </ci:container-interceptors>
    </jee:assembly-descriptor>
</jboss>
```

- **urn:container-interceptors:1.0** ネームスペース用 XSD は https://github.com/jbossas/jboss-as/blob/master/ejb3/src/main/resources/jboss-ejb-container-interceptors_1_0.xsd で入手できます。

バグを報告する

16.4. セキュリティーコンテキスト ID の変更

概要

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 にリモートコールを行う場合は、サーバーへの接続が認証され、この接続を介して受信されたすべての要求が、接続を認証した ID として実行されます。これは、クライアントとサーバー間のコールとサーバー間のコールの両方に適用されます。同じクライアントから異なる ID を使用する必要がある場合は、通常、サーバーに対して複数の接続を開き、各接続が異なる ID として認証されるようにする必要があります。複数のクライアント接続を開く代わりに、認証済みユーザーに別のユーザーとして要求を実行するパーミッションを与えることができます。

このトピックでは、既存のクライアント接続の ID を切り替える方法について説明します。完全な実例については、**ejb-security-interceptors** クイックスタートを参照してください。以下のコード例は、クイックスタートのコードを抜粋したものです。

手順16.2 セキュリティーコンテキストの ID の変更

セキュアな接続の ID を変更するには、以下の 3 つのコンポーネントを作成する必要があります。

1. **クライアントサイドインターセプターの作成**
   このインターセプターは、**org.jboss.ejb.client.EJBClientInterceptor** を実装する必要があります。インターセプターは、コンテキストデータマップを介して要求された ID を渡すことが期待されます。このコンテキストデータマップは、**EJBClientInvocationContext.getContextData()** への呼び出しを介して取得できます。クライアントサイドインターセプターの例は、以下のとおりです。

```
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();
```

```
    }
}
```

ユーザーアプリケーションは、以下のいずれかの方法で **EJBClientContext** のインターセプターに接続できます。

- **プログラミング**
  この方法で
  は、**org.jboss.ejb.client.EJBClientContext.registerInterceptor(int order, EJBClientInterceptor interceptor)** API を呼び出し、**order** および **interceptor** インスタンスを渡します。**order** は、この **interceptor** が置かれるクライアントインターセプターチェーンの位置を決定するために使用されます。

- **ServiceLoader メカニズム**
  この方法では、**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する必要があります。ファイルのルールは、Java ServiceLoader メカニズムにより決まります。このファイルでは、EJB クライアントインターセプター実装の完全修飾名が各行に含まれることが期待されます。EJB クライアントインターセプタークラスがクラスパスで利用可能である必要があります。**ServiceLoader** 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、クライアントインターセプターチェーンの最後に、クラスパスに指定された順序で追加されます。**ejb-security-interceptors** クイックスタートでは、この方法が使用されます。

2. **サーバーサイドコンテナーインターセプターの作成および設定**
   コンテナーインターセプタークラスは、単純な Plain Old Java Object (POJO) です。**@javax.annotation.AroundInvoke** 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指定します。コンテナーインターセプターの詳細については、「コンテナーインターセプターについて」を参照してください。

   a. **コンテナーインターセプターの作成**
      このインターセプターは、ID で **InvocationContext** を受け取り、切り替えを要求します。実際のコード例を抜き出したものは以下のとおりです。

```
    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;
            RealmUser connectionUser = null;

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

```
SecurityActions.remotingContextGetConnection();
                if (con != null) {
                    UserInfo userInfo = con.getUserInfo();
                    if (userInfo instanceof SubjectUserInfo) {
                        SubjectUserInfo sinfo =
(SubjectUserInfo) userInfo;
                        for (Principal current :
sinfo.getPrincipals()) {
                            if (current instanceof RealmUser) {
                                connectionUser = (RealmUser)
current;
                                break;
                            }
                        }
                    }
                } else {
                    throw new IllegalStateException("Delegation
user requested but no user on connection found.");
                }
            }

            SecurityContext cachedSecurityContext = null;
            boolean contextSet = false;
            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 switch user
and have successfully identified the user from the connection
                        // so now we attempt the switch.
                        cachedSecurityContext =
SecurityActions.securityContextSetPrincipalInfo(desiredUser,
                                new
OuterUserCredential(connectionUser));
                        // keep track that we switched the
security context
                        contextSet = true;
                        SecurityActions.remotingContextClear();
                    } 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 security context
                if (contextSet) {
```

```
SecurityActions.securityContextSet(cachedSecurityContext);
                }
            }
        }
    }
```

b. **コンテナーインターセプターの設定**
サーバーサイドコンテナーインターセプターの設定方法については、「コンテナーインターセプターの設定」を参照してください。

3. **JAAS LoginModule の作成**
このコンポーネントは、ユーザが要求された ID として要求を実行することが許可されていることを確認します。以下のコード例は、ログインと検証を実行するメソッドを示しています。

```
@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;
        }
        return false; // If the CallbackHandler can not handle the
required callbacks then no chance.
    }
    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.
    }
```

```
    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[1])) {
            // A wild card mapping was found.
            return true;
        }
        for (String current : allowedMappings) {
            if (requestedUser.equals(current)) {
                return true;
            }
        }
        return false;
    }
```

完全な指示とコードの詳細については、**README** ファイルを参照してください。

バグを報告する

16.5. EJB 認証のために追加セキュリティーを提供する

概要

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 にリモートコールを行う場合は、サーバーへの接続が認証され、この接続を介して受信されたすべての要求が、接続を認証したクレデンシャルを使用して実行されます。接続レベルでの認証は、基礎となる SASL (Simple Authentication and Security Layer) の機能に依存します。カスタム SASL メカニズムを記述する代わりに、サーバーに対する接続を開いて認証し、EJB を呼び出す前にセキュリティートークンを追加できます。このトピックでは、EJB 認証のために既存のクライアント接続で追加情報を渡す方法について説明します。

以下のコード例は、デモ目的専用です。これらのコード例は 1 つの方法のみを示し、アプリケーションのニーズに応じてカスタマイズする必要があります。パスワードは、SASL メカニズムを使用して交換されます。SASL DIGEST-MD5 認証が使用される場合、パスワードはチャンレンジ値でハッシュ化され、平文で送信されません。ただし、残りのトークンは平文で送信されます。これらのトークンに機密情報が含まれる場合は、接続の暗号化を有効にできます。

手順16.3 EJB 認証のためにセキュリティー情報を渡す

認証された接続に追加セキュリティーを提供するには、以下の 3 つのコンポーネントを作成する必要があります。

1. **クライアントサイドインターセプターを作成する**
   このインターセプターは、**`org.jboss.ejb.client.EJBClientInterceptor`** を実装する必要があります。インターセプターは、コンテキストデータマップを介して追加セキュリティートークンを渡すことが期待されます。このコンテキストデータマップは、**`EJBClientInvocationContext.getContextData()`** への呼び出しを介して取得できます。追加セキュリティートークンを作成するクライアントサイドインターセプターコードの例は、以下のとおりです。

```
public class ClientSecurityInterceptor implements
EJBClientInterceptor {

    public void handleInvocation(EJBClientInvocationContext context)
throws Exception {
        Object credential =
SecurityActions.securityContextGetCredential();

        if (credential != null && credential instanceof
PasswordPlusCredential) {
            PasswordPlusCredential ppCredential =
(PasswordPlusCredential) credential;
            Map<String, Object> contextData =
context.getContextData();

contextData.put(ServerSecurityInterceptor.SECURITY_TOKEN_KEY,
                                ppCredential.getAuthToken());
        }
        context.sendRequest();
    }

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

   クライアントインターセプターをアプリケーションに接続する方法については、「アプリケーションでのクライアントサイドインターセプターの使用」を参照してください。

2. **サーバーサイドコンテナーインターセプターを作成および設定する**
   コンテナーインターセプタークラスは、単純な Plain Old Java Object (POJO) です。**`@javax.annotation.AroundInvoke`** 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指定します。コンテナーインターセプターの詳細については、「コンテナーインターセプターについて」を参照してください。

   a. **コンテナーインターセプターを作成する**
      このインターセプターは、コンテキストからセキュリティー認証トークンを取得し、認証のために JAAS (Java Authentication and Authorization Service) ドメインに渡します。コンテナーインターセプターコードの例は以下のとおりです。

```
public class ServerSecurityInterceptor {
```

```
    private static final Logger logger =
Logger.getLogger(ServerSecurityInterceptor.class);
    static final String SECURITY_TOKEN_KEY =
ServerSecurityInterceptor.class.getName() + ".SecurityToken";

    @AroundInvoke
    public Object aroundInvoke(final InvocationContext
invocationContext) throws Exception {
        Principal userPrincipal = null;
        RealmUser connectionUser = null;
        String authToken = null;

        Map<String, Object> contextData =
invocationContext.getContextData();
        if (contextData.containsKey(SECURITY_TOKEN_KEY)) {
            authToken = (String)
contextData.get(SECURITY_TOKEN_KEY);

            Connection con =
SecurityActions.remotingContextGetConnection();

            if (con != null) {
                UserInfo userInfo = con.getUserInfo();
                if (userInfo instanceof SubjectUserInfo) {
                    SubjectUserInfo sinfo = (SubjectUserInfo)
userInfo;
                    for (Principal current :
sinfo.getPrincipals()) {
                        if (current instanceof RealmUser) {
                            connectionUser = (RealmUser)
current;
                            break;
                        }
                    }
                }
                userPrincipal = new
SimplePrincipal(connectionUser.getName());

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

        SecurityContext cachedSecurityContext = null;
        boolean contextSet = false;
        try {
            if (userPrincipal != null && connectionUser != null
&& authToken != null) {
                try {
                    // We have been requested to use an
authentication token
                    // so now we attempt the switch.
                    cachedSecurityContext =
SecurityActions.securityContextSetPrincipalCredential(userPrincip
al,
```

```
                                new
OuterUserPlusCredential(connectionUser, authToken));
                        // keep track that we switched the security
context
                        contextSet = true;
                        SecurityActions.remotingContextClear();
                    } 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 security context
                if (contextSet) {

SecurityActions.securityContextSet(cachedSecurityContext);
                }
            }
        }
}
```

b. **コンテナーインターセプターを設定する**
サーバーサイドコンテナーインターセプターの設定方法については、「コンテナーインターセプターの設定」を参照してください。

3. **JAAS LoginModule を作成する**
このカスタムモジュールは、既存の認証済み接続情報と追加セキュリティートークンを使用して認証を実行します。追加セキュリティートークンを使用し、認証を実行するコードの例は以下のとおりです。

```
public class SaslPlusLoginModule extends AbstractServerLoginModule {

    private static final String ADDITIONAL_SECRET_PROPERTIES =
"additionalSecretProperties";
    private static final String DEFAULT_AS_PROPERTIES = "additional-
secret.properties";
    private Properties additionalSecrets;
    private Principal identity;

    @Override
    public void initialize(Subject subject, CallbackHandler
callbackHandler, Map<String, ?> sharedState, Map<String, ?> options)
{
        addValidOptions(new String[] { ADDITIONAL_SECRET_PROPERTIES
});
        super.initialize(subject, callbackHandler, sharedState,
options);

        // Load the properties that contain the additional security
```

```
tokens
        String propertiesName;
        if (options.containsKey(ADDITIONAL_SECRET_PROPERTIES)) {
            propertiesName = (String)
options.get(ADDITIONAL_SECRET_PROPERTIES);
        } else {
            propertiesName = DEFAULT_AS_PROPERTIES;
        }
        try {
            additionalSecrets =
SecurityActions.loadProperties(propertiesName);
        } catch (IOException e) {
            throw new
IllegalArgumentException(String.format("Unable to load properties
'%s'", propertiesName), e);
        }
    }

    @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;
            }
            return false; // If the CallbackHandler can not handle
the required callbacks then no chance.
        }

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

        if (credential instanceof OuterUserPlusCredential) {
            OuterUserPlusCredential oupc = (OuterUserPlusCredential)
credential;
            if (verify(name, oupc.getName(), oupc.getAuthToken())) {
                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", oupc);
                }
                loginOk = true;
                return true;
            }
        }

        return false; // Attempted login but not successful.
    }

    private boolean verify(final String authName, final String
connectionUser, final String authToken) {
        // For the purpose of this quick start we are not supporting
switching users, this login module is validation an
        // additional security token for a user that has already
passed the sasl process.
        return authName.equals(connectionUser) &&
authToken.equals(additionalSecrets.getProperty(authName));
    }

    @Override
    protected Principal getIdentity() {
        return identity;
    }

    @Override
    protected Group[] getRoleSets() throws LoginException {
        Group roles = new SimpleGroup("Roles");
        Group callerPrincipal = new SimpleGroup("CallerPrincipal");
        Group[] groups = { roles, callerPrincipal };
        callerPrincipal.addMember(getIdentity());
        return groups;
    }
}
```

4. **カスタム LoginModule をチェーンに追加する**
   新しいカスタム LoginModule はチェーンの正しい場所に追加して正しい順序で呼び出されるようにする必要があります。この例では、**SaslPlusLoginModule** は、**password-stacking** オプションセットでロールをロードする LoginModule の前にチェーンする必要があります。

   - **管理 CLI を使用して LoginModule 順序を設定する**
     password-stacking オプションを設定する **RealmDirect** LoginModule の前にカスタム **SaslPlusLoginModule** をチェーンする管理 CLI コマンドの例は以下のとおりです。

```
[standalone@localhost:9999 /] ./subsystem=security/security-
domain=quickstart-domain:add(cache-
type=default)[standalone@localhost:9999 /]
./subsystem=security/security-domain=quickstart-
domain/authentication=classic:add[standalone@localhost:9999 /]
./subsystem=security/security-domain=quickstart-
domain/authentication=classic/login-
module=DelegationLoginModule:add(code=org.jboss.as.quickstarts.ej
b_security_plus.SaslPlusLoginModule,flag=optional,module-options=
{password-stacking=useFirstPass})[standalone@localhost:9999 /]
```

```
./subsystem=security/security-domain=quickstart-
domain/authentication=classic/login-
module=RealmDirect:add(code=RealmDirect,flag=required,module-
options={password-stacking=useFirstPass})
```

管理 CLI の詳細については、カスタマーポータル (https://access.redhat.com/site/documentation/JBoss_Enterprise_Application_Platform/) にある JBoss Enterprise Application Platform 6 向け 『管理および設定ガイド』 の章 「『管理インターフェース』」を参照してください。

- **LoginModule 順序を手動で設定する**
  以下に、サーバー設定ファイルの **security** サブシステムで LoginModule 順序を設定する XML の例を示します。カスタム **SaslPlusLoginModule** は **RealmDirect** LoginModule より前に指定してユーザーロールがロードされ、**password-stacking** オプションが設定される前にリモートユーザーを確認できるようにする必要があります。

```
<security-domain name="quickstart-domain" cache-type="default">
    <authentication>
        <login-module
code="org.jboss.as.quickstarts.ejb_security_plus.SaslPlusLoginMod
ule" flag="required">
            <module-option name="password-stacking"
value="useFirstPass"/>
        </login-module>
        <login-module code="RealmDirect" flag="required">
            <module-option name="password-stacking"
value="useFirstPass"/>
        </login-module>
    </authentication>
</security-domain>
```

5. **リモートクライアントを作成する**
   以下のコード例では、上記の JAAS LoginModule によりアクセスされる **additional-secret.properties** ファイルに以下のプロパティーが含まれることを前提とします。

```
quickstartUser=7f5cc521-5061-4a5b-b814-bdc37f021acc
```

以下のコードは、EJB 呼び出しの前にセキュリティートークンを作成し、設定する方法を示しています。シークレットトークンはデモ目的のためにのみハードコーディングされています。このクライアントは、単に結果をコンソールに出力します。

```
import static
org.jboss.as.quickstarts.ejb_security_plus.EJBUtil.lookupSecuredEJB;

public class RemoteClient {

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
```

```
        SimplePrincipal principal = new
SimplePrincipal("quickstartUser");
        Object credential = new
PasswordPlusCredential("quickstartPwd1!".toCharArray(), "7f5cc521-
5061-4a5b-b814-bdc37f021acc");


SecurityActions.securityContextSetPrincipalCredential(principal,
credential);
        SecuredEJBRemote secured = lookupSecuredEJB();

        System.out.println(secured.getPrincipalInformation());
    }
}
```

バグを報告する

16.6. アプリケーションでのクライアントサイドインターセプターの使用

概要

プログラミングまたは ServiceLoader メカニズムを使用して、クライアントサイドインターセプターをアプリケーションに接続できます。この 2 つの方法の詳細は以下のとおりです。

手順16.4 インターセプターを接続する

- **プログラミング**
  この方法で
  は、**`org.jboss.ejb.client.EJBClientContext.registerInterceptor(int order, EJBClientInterceptor interceptor)`** API を呼び出し、**`order`** および **`interceptor`** インスタンスを渡します。**`order`** は、この **`interceptor`** が置かれるクライアントインターセプターチェーンの位置を決定するために使用されます。

- **ServiceLoader メカニズム**
  この方法では、**`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する必要があります。ファイルのルールは、Java ServiceLoader メカニズムにより決まります。このファイルでは、EJB クライアントインターセプター実装の完全修飾名が各行に含まれることが期待されます。EJB クライアントインターセプタークラスがクラスパスで利用可能である必要があります。**`ServiceLoader`** 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、クライアントインターセプターチェーンの最後に、クラスパスに指定された順序で追加されます。

バグを報告する

第17章 開発セキュリティーに関する参考資料

17.1. JBOSS-WEB.XML の設定に関する参考資料

はじめに

jboss-web.xml はデプロイメントの**WEB-INF** または **META-INF** ディレクトリ内にあるファイルです。このファイルには、JBoss Web コンテナが Servlet 3.0 仕様に追加する機能に関する設定情報が含まれています。Servlet 3.0 仕様は **web.xml** の同じディレクトリに格納されます。

jboss-web.xml ファイルのトップレベル要素は **<jboss-web>** 要素です。

グローバルリソースの WAR 要件へのマッピング

使用可能な設定の多くは、アプリケーションの **web.ml** に設定される要件をローカルリソースへマッピングします。**web.xml** の設定に関する説明は http://docs.oracle.com/cd/E13222_01/wls/docs81/webapp/web_xml.html を参照してください。

例えば、**web.xml** に **jdbc/MyDataSource** が必要な場合、**jboss-web.xml** はグローバルデータソース **java:/DefaultDS** をマッピングして要件を満たすことがあります。WAR はグローバルデータソースを使用して **jdbc/MyDataSource** に対する要求を満たします。

表17.1 一般的なトップレベル属性

属性	説明
env-entry	**web.xml** が必要とする **env-entry** へのマッピング。
ejb-ref	**web.xml** が必要とする **ejb-ref** へのマッピング。
ejb-local-ref	**web.xml** が必要とする **ejb-local-ref** へのマッピング。
service-ref	**web.xml** が必要とする **service-ref** へのマッピング。
resource-ref	**web.xml** が必要とする **resource-ref** へのマッピング。
resource-env-ref	**web.xml** が必要とする**resource-env-ref** へのマッピング。
message-destination-ref	**web.xml** が必要とする **message-destination-ref** へのマッピング。
persistence-context-ref	**web.xml** が必要とする **persistence-context-ref** へのマッピング。
persistence-unit-ref	**web.xml** が必要とする **persistence-unit-ref** へのマッピング。

属性	説明
post-construct	**web.xml** が必要とする **post-context** へのマッピング。
pre-destroy	**web.xml** が必要とする **pre-destroy** へのマッピング。
data-source	**web.xml** が必要とする **data-source** へのマッピング。
context-root	アプリケーションのルートコンテキスト。デフォルト値は **.war** サフィックスを除いたデプロイメントの名前です。
virtual-host	アプリケーションがリクエストを許可する HTTP 仮想ホストの名前。HTTP の **Host** ヘッダーの内容を参照します。
annotation	アプリケーションによって使用されるアノテーションを記述します。詳細は <annotation> を参照してください。
listener	アプリケーションによって使用されるリスナーを記述します。詳細は <listener> を参照してください。
session-config	この要素は **web.xml** の **<session-config>** 要素と同じ関数を入力します。互換性維持の目的でのみ含まれます。
valve	アプリケーションによって使用されるバルブを記述します。詳細は <valve> を参照してください。
overlay	アプリケーションに追加するオーバーレイの名前。
security-domain	アプリケーションによって使用されるセキュリティードメインの名前。セキュリティードメイン自体は Web ベースの管理コンソールか管理 CLI に設定されます。
security-role	この要素は **web.xml** の **<security-role>** 要素と同じ関数を入力します。互換性維持の目的でのみ含まれます。

属性	説明
use-jboss-authorization	この要素が存在し、大文字と小文字を区別しない true という値が含まれる場合、JBoss Web 承認スタックが使用されます。この要素が存在しない場合や、true でない値が含まれる場合は、Java enterprise Edition 仕様に指定された承認メカニズムのみが使用されます。この要素は JBoss Enterprise Application Platform 6 に新規導入された要素です。
disable-audit	この空の要素が存在する場合、Web セキュリティー監査が無効になります。Web セキュリティー監査は Java EE 仕様の一部ではありません。この要素は JBoss Enterprise Application Platform 6 に初めて導入された要素です。
disable-cross-context	**false** の場合、アプリケーションは他のアプリケーションコンテキストを呼び出すことができます。デフォルトは **true** です。

以下の各要素は子要素を持っています。

<annotation>

アプリケーションによって使用されるアノテーションを記述します。下表は **<annotation>** の子要素の一覧になります。

表17.2 アノテーション設定要素

属性	説明
class-name	アノテーションのクラスの名前。
servlet-security	サーブレットのセキュリティーを表す **@ServletSecurity** などの要素。
run-as	run-as の情報を表す **@RunAs** などの要素。
multi-part	マルチパートの情報を表す **@MultiPart** などの要素。

<listener>

リスナーを記述します。下表は **<listener>** の子要素の一覧になります。

表17.3 リスナー設定要素

属性	説明
class-name	リスナーのクラスの名前。

属性	説明
listener-type	アプリケーションのコンテキストにどのようなリスナーを追加するかを示す **condition** 要素の一覧です。以下を選択することが可能です。 **CONTAINER** コンテキストに ContainerListener を追加します。 **LIFECYCLE** コンテキストに LifecycleListener を追加します。 **SERVLET_INSTANCE** コンテキストに InstanceListener を追加します。 **SERVLET_CONTAINER** コンテキストに WrapperListener を追加します。 **SERVLET_LIFECYCLE** コンテキストに WrapperLifecycle を追加します。
module	リスナークラスが含まれるモジュールの名前。
param	パラメーター。**<param-name>** と **<param-value>** の 2 つの子要素が含まれます。

<valve>

アプリケーションのバルブを記述します。<listener> と同じ設定要素が含まれます。

バグを報告する

17.2. EJB セキュリティーパラメーターについての参考資料

表17.4 EJB セキュリティーパラメーター要素

要素	説明
<security-identity>	EJB のセキュリティー ID に付随する子要素が含まれています。
<use-caller-identity />	EJB が呼び出し側と同じセキュリティー ID を使うよう指定します。
<run-as>	**<role-name>** 要素が含まれています。

要素	説明
<run-as-principal>	存在する場合、発信呼び出しへ割り当てられたプリンシパルを示します。存在しない場合、発信呼び出しは **anonymous** という名前のプリンシパルへ割り当てられます。
<role-name>	EJB が実行されるロールを指定します。
<description>	**<role-name>** に名前のあるロールを記述します。

例17.1 セキュリティー ID の例

この例は、表17.4「EJB セキュリティーパラメーター要素」 に説明のある各タグを示しています。これらのタグは**<session>** の中でのみ利用可能です。

```
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>ASessionBean</ejb-name>
            <security-identity>
                <use-caller-identity/>
            </security-identity>
        </session>
        <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>
    <session>
    <ejb-name>RunAsBean</ejb-name>
    <security-identity>
    <run-as-principal>internal</run-as-principal>
    </security-identity>
    </session>
    </enterprise-beans>
</ejb-jar>
```

バグを報告する

第18章 補足参考資料

18.1. JAVA ARCHIVEの種類

JBoss Enterprise Application Platform は様々な種類のアーカイブファイルを認識します。アーカイブファイルは、デプロイ可能なサービスとアプリケーションをパッケージ化するために使用されます。

一般的に、アーカイブファイルは特定のファイル拡張とディレクトリ構造を持つ zip アーカイブです。Zip アーカイブがアプリケーションサーバーにデプロイされる前に展開されると、展開済みアーカイブとして参照されます。その場合、ディレクトリ名にはファイルの拡張子が含まれており、ディレクトリ構造の要件も適用されます。

表18.1

アーカイブタイプ	拡張	目的	ディレクトリ構造の要件
Java アーカイブ	.jar	Java クラスのライブラリが含まれています。	**`META-INF/MANIFEST.MF`** ファイル (オプション)。どのクラスが **`main`** クラスであるかなどの情報を指定します。
Web アーカイブ	.war	Java クラスおよびライブラリ以外に、Java Server Pages (JSP) ファイル、サーブレット、および XML ファイルが含まれています。Web アーカイブのコンテンツは Web アプリケーションとも呼ばれます。	**`WEB-INF/web.xml`** ファイル。Web アプリケーションの構造に関する情報が含まれています。**`WEB-INF/`** には、他のファイルが存在する場合もあります。
リソースアダプターアーカイブ	.rar	ディレクトリ構造は、JCA 仕様で指定されています。	Java Connector Architecture (JCA) リソースアダプターが含まれています。コネクターとも呼ばれます。
エンタープライズアーカイブ	.ear	1 つ以上のモジュールを 1 つのアーカイブにパッケージ化してそれらのモジュールをアプリケーションサーバーに同時にデプロイできるようにするために Java Enterprise Edition (EE) によって使用されます。EAR アーカイブを構築するツールで最も一般的なものは Maven および Ant です。	**`META-INF/`** ディレクトリ。このディレクトリには 1つ以上の XML デプロイメント記述子ファイルが含まれています。

アーカイブタイプ	拡張	目的	ディレクトリ構造の要件
			以下のモジュールタイプのいずれか • Web アーカイブ (WAR) • Plain Old Java Object (POJO) を含む Java Archive (JAR) 1 つ以上 • 独自の **META-INF/** ディレクトリを含むエンタープライズ JavaBean (EJB) モジュール 1 つ以上。このディレクトリには、デプロイされる永続クラスの記述子が含まれています。 • リソースアーカイブ (RAR) 1 つ以上
サービスアーカイブ	.sar	エンタープライズアーカイブに類似しますが、JBoss Enterprise Application Platform に固有なものです。	**jboss-service.xml** または **jboss-beans.xml** ファイルを含む **META-INF/** ディレクトリ。

バグを報告する

付録A REVISION HISTORY

改訂 1.1-2 **Thu Jul 11 2013** **Russell Dickenson**

JBoss Enterprise Application Platform 6.1.0 GA Release.